ECTI Transactions on Computer and Information Technology
Journal homepage: https://ph01.tci-thaijo.org/index.php/ecticit/
Published by the ECTI Association, Thailand, ISSN: 2286-9131

Codepickle: Portable Python Grid Computing

Eakasit Tangmunchittham[1] and Krerk Piromsopa[2]

ABSTRACT

We present Codepickle, a groundbreaking solution designed to overcome Python version constraints and enhance portability, especially within distributed Python grids and volunteer computing environments. Unlike existing serialization libraries such as Cloudpickle and Dill, which rely on Python bytecode and are prone to version conflicts, Codepickle offers a robust alternative. Our methodology includes innovative adjustments for function serialization and shared variable management. Experimental results reveal challenges like code sourcing and nonlocal variable handling. Performance benchmarks highlight Codepickle's significant advantages over Cloudpickle, including better portability and reduced message sizes. Notably, Codepickle achieves message sizes that are 84% of those produced by Cloudpickle especially for small code segments, with comparable execution performance. Proposed enhancements target critical issues such as lambda functions and cross-version compatibility. This comprehensive study not only demonstrates Codepickle's transformative potential but also underscores the ongoing quest for advanced serialization techniques in Python's distributed computing landscape.

Article information:

Keywords: Distributed System, Python, Serialization, Computer Architecture

Article history:
Received: May 22, 2024
Revised: July 25, 2024
Accepted: September 19, 2024
Published: October 19, 2024 (Online)

DOI: 10.37936/ecti-cit.2024184.256835

1. INTRODUCTION

Notably, Python has emerged as an indispensable tool in the field of AI and data analytics [1], [2], [3], [4], [5]. Its importance is rooted in its user-friendly nature, extensive library support, and cross-platform compatibility. As the computing power of individual computers is limited, there has been research and development of distributed computing frameworks to address this limitation. Currently, Python has a wide range of frameworks that support distributed systems in the form of clusters, such as Ray [6], pathos [7], PySpark [8], ipyparallel [9], and joblib [10]. In these setups, effective communication hinges on the choice of messaging patterns which can cover several layers similar to Network OSI [11] pattern. The serialization serves as a critical process for converting between objects and messages.

The operation diverges significantly from the JavaScript client-server architecture, which relies on sending code for browser interpretation and exchanging messages primarily in plaintext, CSV [12], JSON [13], or XML [14] formats. Python, on the other hand, facilitates the transmission of data as objects, encompassing functions, classes, and operation results. This approach includes all related dependencies and object properties, thereby minimizing the need for custom message parsers. While this concept extends beyond Python, its most prominent implementations are found within the Python ecosystem. Notably, the libraries Cloudpickle [15] and Dill [16] exemplify this approach. These libraries, which share similar underlying mechanisms, are integral to Python's distributed and clustered frameworks, serving as both default and configurable options for efficient object serialization.

We explore opportunities to expand the scope from clusters to grids [17] and voluntary computing [18]. This expansion would allow for the larger number of computation nodes for some algorithms such as Embarrassingly Parallel [19], [20]. Several limitations are Security [21], [22] and Portability [23]. We omitted the analysis on the relationship between models, algorithms and architectures as they have been covered by other researches [24].

As for the security, these distributed execution frameworks are designed to perform remote code ex-

[1,2]The authors are with the Department of Computer Engineering, Faculty of Engineering, Chulalongkorn University, Bangkok, Thailand, E-mail: eakasit.tangmun@gmail.com and krerk.p@chula.ac.th
[2]Corresponding author: krerk.p@chula.ac.th

ecution (RCE) as-a service. So both grid and voluntary computing will require the setup of external Python sandbox separately. This can be done either at the Operating System level or at the Browser level such as Pyodide [25].

As previously analyzed in the paper [26], portability aligns closely with version compatibility and, for ease of analysis, can be subdivided into multiple levels. We acknowledge that both Cloudpickle and Dill have inherent the same limitations. They rely on Python bytecode and function constructors to support objects like functions, classes, and lambda functions. Since Python bytecode is a low-level implementation susceptible to changes in different minor Python versions, both Cloudpickle and Dill require that serialization and deserialization occur within the exact same Python version. While this requirement aligns well in cluster computing environments, it becomes a challenge in loosely distributed systems such as grids or volunteer computing, where individual computers may vary in terms of heterogeneity and computing setups. This raises the critical question whether there is a viable solution to improve and to eliminate these versioning restrictions.

Our contributions are in four main parts:

- Explore an alternative method for serializing function and class objects that provides a workaround to the Python version limitation.
- Provide an in-depth explanation of the internal processes and enumerate the potential trade-offs and limitations inherent to Python. These limitations become evident as we showcase them through our Codepickle module, which is demonstrated by conducting a comprehensive evaluation using the original Cloudpickle testsuite.
- Evaluate the performance against other methods.
- Introduce a method for extracting crucial module import information from serialized objects. This process serves as a valuable tool for configuring the computing environment on distributed nodes efficiently.

While our implementation is based on Python pickle, there is no theoretical limitation to apply the concept to other interpreted languages. Through the integration of Codepickle into established frameworks, we have enabled the seamless execution of Python code across a diverse grid infrastructure, supporting various Python versions. Our experimentation, adapting from standard Python tests found in CPython [27] repository as a benchmark, has illustrated that with an increasing number of nodes, the benefits provided by Codepickle significantly outweigh any potential drawbacks.

The structure of the remaining sections in this paper is outlined as follows: Section 2 provides the necessary background to comprehend functions, objects, and serialization. Section 3 explains test suites. Section 4 expounds on our design and methodology. Sections 5 and 6 delve into performance analysis and our insights for improvement. In Section 7, we conduct a compatibility analysis. Lastly, Section 8 is dedicated to our discussions and conclusions.

2. BACKGROUND

2.1 Serialization

In object-oriented programming, serialization serves as a fundamental mechanism for converting an object's state into a transmittable and reconstructable message format. This process enables the persistence and exchange of complex data structures across diverse systems and storage solutions.

A rich ecosystem of serialization standards has emerged, each optimized for specific use cases and data types. Common examples include JSON (JavaScript Object Notation), XML (Extensible Markup Language), Java Serialization [28], PHP Serialization [29], and Pickle (Python) [30]. While some standards prioritize simplicity by supporting fundamental data types like integers, strings, arrays, and dictionaries, others offer more advanced functionalities. These advanced features encompass the ability to handle references, circular references, and even user-defined objects and classes.

Cloudpickle exemplifies such an advanced serialization standard. Building upon Pickle's capabilities, Cloudpickle extends its functionality to serialize not only class and module references but also the underlying data of functions and classes themselves. This enhanced capacity empowers developers with a robust mechanism for data exchange and persistence in complex programming environments. Internally, Cloudpickle accomplishes this by creating a function constructor that accepts the function's bytecode and properties as an input message, and outputs the function object to a process.

2.2 Function Objects

A function in programming is a callable unit of software logic with a well-defined interface and behavior, designed to be invoked multiple times. Fundamentally, a function is a collection of data that includes instructions directing the processor to execute specific commands. In low-level languages, such as assembly, these instruction sets are highly dependent on the platform and architecture. Conversely, in high-level languages, the instructions may be processed through additional abstraction layers, such as a stack machine, rendering them more platform independent. In the latter case, it becomes feasible to transfer a function from one processing node to another through communication channels.

In Python, a function is treated as an object, similar to other data types, and the language's design allows programmers to access its internal properties. Key attributes of a function object include '__code__', '__globals__', and '__closure__' The Cloudpickle module leverages these attributes extensively, transmitting them to another node and using them to reconstruct the function from scratch. The feasibility of this process hinges on the language's API for accessing the internal properties of function objects, as well as the API for reconstructing these functions. While a similar approach could potentially be applied in other programming languages, it remains beyond the scope of the current research.

2.3 Directly Passing Source Code vs. Serialization

The choice between directly passing source code and serialization has a profound impact on API design and application development. Selecting the right approach fosters a user friendly and maintainable system. Directly passing source code entrusts users with crafting the message for transmission. This approach, however, can introduce security vulnerabilities, increase user complexity, and make code consistency challenging to maintain.

Serialization, in contrast, acts as a higher-level abstraction, effortlessly converting objects into transmittable messages. This approach fosters seamless collaboration between the main program and distributed framework functions within the same layer. Invoking the distributed framework API becomes as straightforward as making a function call with the necessary arguments. Additionally, serialization often offers platform independence, facilitates versioning (if applicable), and potentially allows for performance optimizations.

2.4 Usefulness of Extracting Imported Modules Information

Loosely distributed systems, like grid and voluntary computing, face unique challenges due to the dynamic nature of participating nodes. These nodes may be set up at varying times or only during connection initiation, making it difficult to guarantee a consistent environment. In Python, applications often rely on external modules and packages for specific functionalities.

3. TEST SUITES

Given the potential impact of the Codepickle module on various areas, including enhanced functionality and improved portability, several considerations must be addressed. To ensure comprehensive coverage, we have meticulously planned and defined each test suite, detailing the preparation and evaluation methods, as well as the specific objectives for each test.

The kernel test suite serves as the functional unit tests for the Codepickle module. While there is a potential for incompleteness in this approach, its advantages are significant. The suite's simplicity and isolation from the broader Cloudpickle test suite enable us to more effectively analyze the module's behavior and prevent overfitting to the specificities of the Cloudpickle tests, as discussed in Section 3.2. Comprehensive testing of Python syntax and functionality is covered in Section 3.3.

3.1 Kernel Test Suite

This unit test is well-defined set of function objects selected from both the main Python and famous developer documentations. Below is the list of test cases included in our kernel test suite:

- Functions: 2 tests
- Functions calling another function: 1 test
- Functions with inner function: 1 test
- Functions using global variables: 2 tests
- Recursive Functions: 2 tests
- User-defined class: 2 tests
  - Enum-class: 1 test
  - ABC-class: 1 test
  - Super-class: 1 test
  - Inherited-class: 2 tests
  - Class instance: 1 test
- Imported modules: 3 tests
- Lambda function: 4 tests
- Function with decorator: 4 tests
- Function with closure variables: 6 tests
- Class with decorator: 1 test
- Function with type hint: 1 test

Although our initial focus may not cover every possible function object, any gap will be thoroughly addressed during the final testing phase, which employs the complete Cloudpickle test suite.

3.2 Cloudpickle Test Suite

Our Cloudpickle test suite comprises 128 tests and serves as a comprehensive functional comparison against other serialization modules. We directly adopted the test suite from the Cloudpickle GitHub repository [15], making only one modification—replacing the calls to Cloudpickle with Codepickle—without altering any other aspects of the tests. The Cloudpickle test suite includes extensive use of subprocess calls, the 'exec' function, multiple dumps and loads within single tests, and lambda functions within its framework. Consequently, when the Codepickle module fails to handle lambda functions, it also fails other functional tests that indirectly utilize lambdas. Rather than altering the original test code, we opted to conduct a thorough analysis and classification of all failure cases.

3.3 Bytecode Compatibility Test Suite

Unlike functionality tests, our 152 bytecode compatibility tests are specifically designed to verify that our module can sustain portability across different Python versions. Cloudpickle's dependency on

Python bytecode presents substantial challenges in maintaining portability between versions, impacting both major and minor updates. A comprehensive overview of the changes in Python bytecode across versions is documented in [33]. Failure to adhere to this compatibility requirement can lead to numerous issues, including Python exceptions caused by structural differences in functions detected by the Cloudpickle process, or even cPython segmentation faults.

In contrast to Cloudpickle, which relies on Python bytecode, Codepickle utilizes Python source code for function object representation. This approach capitalizes on the well-established portability of Python code within the same major version (e.g., Python 3.x). To evaluate Codepickle's compatibility across Python versions 3.7 to 3.11, we conducted a focused analysis, identifying frequent changes in bytecode across these versions. For example, Bytecode No. 152, used for unpacking tuples in Python 3.7, was removed in Python 3.9 and then reintroduced in Python 3.10 for the "match" syntax—an operation entirely unrelated to tuple unpacking. In practice, Python bytecode evolves even more frequently in minor versions, but for our research, we selected five key major versions, each with the most up-to-date minor version: 3.7.17, 3.8.17, 3.9.17, 3.10.12, and 3.11.4. We subsequently developed a test suite that includes functions designed to account for these bytecode variations. The complete test suite is available in our repository (Appendix A).

4. METHODOLOGY

Leveraging the foundation provided by the existing pickler system, we present a novel approach to function object serialization that complements, rather than replicates, Cloudpickle's functionalities. Our design prioritizes modularity, with each object type's serialization handled independently. While the focus here remains on core concepts, the complete code, including variable assignments, is available for reference in the appendix (Appendix A).

1) Our aim is to modify only a specific and limited portion of the code, focusing on functions and bytecode serialization. Other object types remain unaffected and unchanged.

2) In the serialization of functions, the source code is obtained using the 'inspect.getsource' function. The iterating serialization mechanism remains unchanged.

3) During deserialization, the function construction is replaced with executing the source code using exec. The structure involving argument values and return values remains consistent with the previous implementation.

4) To manage shared global variables, we harnessed the reference sharing feature of the pickler ('memo'), transmitting the '__globals__' dictionary from serialization to be employed as an 'exec' argument during deserialization. This '__globals__' is not an actual globals scope in main program and is only shared between functions that are originally in the same scope during serialization.

a) We restructured Cloudpickle's '_function_getnewargs' into '_function_get_shared_globals', specifically addressing the base_globals aspect. The relocation of the closure variables part into '_function_getstate' will be discussed in the subsequent point.

5) In case of closure variables, we cannot retain their behavior due to the change of code scope. In other words, when parsing the code from inner function, it will be regarded as a standalone function. We decide to move closure variables into '__globals__' dictionary instead.

6) For imported modules listing, we added few code to perform this process at the end of function '_function_getstate'.

To facilitate a clear comparison between Cloudpickle and our proposed approach, we present the pseudo-code difference for serialization in Code Listing 1 and the pseudo-code difference for deserialization in Code Listing 2. Additionally, for reference, the pseudo-code for the imported modules listing function is provided in Code Listing 3.

The 'inspect.getsource' function from Python's standard library 'inspect' module retrieves the source code of a given object. It achieves this by introspecting the object's attributes, particularly those related to the file and line where the object was defined. While this method offers a convenient way to access source code within the Python scope, it has limitations that will be discussed in a later section.

Code Listing 1: diff of pseudocode of cloudpickle and codepickle during serialize

```
function function_serialize(func):
<<<<<<< Codepickle
+     # Get function source code
+     func_name = get_name(func)
+     source_code = get_source_code(func)
+     reindent_source_code(source_code)
+     # Get shared variables (such as globals)
+     shared_vars = get_shared_var(func)
+     reduce_group = (make_function, (
    source_code, func_name, shared_vars))
=======
-     # Get function bytecodes
-     bytecode, shared_vars, closure_cell =
    function_get_argument(func)
-     reduce_group = (make_function, (bytecode
    , shared_vars, closure_cell))
>>>>>>> Cloudpickle

    # Same for both modules
    obj_property, closure_vars = get_property(
        func)
    setstate_group = (function_setstate, (
        obj_property, closure_vars))

    # Add any extra serialized data such as
        dependencies here
    import_module_list = get_imported_modules(
        func)
    return reduce_group, setstate_group,
        import_module_list
```

Code Listing 2: diff of pseudocode of cloudpickle and codepickle during deserialize

```
<<<<<<< Codepickle
+ function make_function(source_code,
   shared_global):
+     # Python builtin exec method
+     exec(source_code, shared_global)
+     return shared_global[func_name]
=======
- function make_function(bytecode, shared_vars
     , closure_cell):
-     func = type.Function(bytecode,
   shared_vars, closure_cell)
-     return func
>>>>>>> Cloudpickle

function function_setstate(func, obj_property,
     closure_vars):
   set_object_property(func, obj_property)
<<<<<<< Codepickle
+     func.set_global_vars(closure_vars)
=======
-     func.set_closure_vars(closure_vars)
>>>>>>> Cloudpickle
   return func
```

Code Listing 3: codepickle: imported modules listing

```
for var in both(global_vars, closure_vars):
  if type(var) == Module:
    module_name = var.name.split(".")[0]
    if module_name not builtin:
      module_lists.add(module_name)
```

4.1 Preliminary Experiments and Limitations

To assess the effectiveness of our proposed approach, we integrated it into a forked repository of Cloudpickle v2.2.1 and conducted experiments using Python 3.11. Following module implementation, we evaluated the design through two testing sets: a custom test suite and the existing Cloudpickle test suite.

Our custom test suite leveraged the methodology previously outlined by Tangmunchittham and Piromsopa [26], which separates process and test files for serialization and deserialization. This approach is particularly advantageous when testing involves global variables. Traditional testing frameworks often execute tests within the same file, potentially leading to issues with pre-declared global variables masking any problems related to missing information during deserialization.

This evaluation process revealed several limitations associated with our design, which will be discussed in details. The reader may follow the examples for comparison between Cloudpickle and Codepickle of their internal processes and the outcomes in the figure 1.

1) Retrieving the source code using 'inspect.getsource' does not consistently yield the desired output. The challenges and considerations in this process are as follows:

a) Unable to get the source code

i) Certain classes or function objects, constructed from metaclasses in the CPython domain (e.g., collections.namedtuple and dataclasses), may contain runtime-created methods like __new__. Although these objects have complete bytecode, there is no source code.

ii) Functions or classes constructed from eval or exec will lack source code. The workaround will be discussed in section VI.

iii) Some interpreter environments may not support the 'inspect.getsource' feature. Examples include the standard Python interpreter, Pyodide (CPython on the browser), and Jupyter, which supports function source code but not inherited classes. iPython works well for both functions and classes.

b) Output that may need to be modified.

i) In certain circumstances, the output from 'inspect.getsource' may include code unrelated to the serializing object, necessitating removal. For instance, one-line lambdas in the return part, such as 'return lambda x: x+1', are not executable until the extraneous return is removed.

2) Complex Non-Local Variables: Global and Nonlocal

a) Nonlocal syntax and variables pose challenges in most circumstances. As mentioned earlier, deserialized function will turn an inner function into a standalone function which makes 'nonlocal' syntax invalid. This contrasts with the bytecode approach in cloudpickle, where the bytecode remains closure-based and can be bound with the variables later.

b) Even without 'nonlocal' syntax, complex closure functions with the exact same variable names as global variables may yield incorrect runtime results. As mentioned earlier, this only applies to the global variables of the objects within the same serialization process, not the actual program's global variables.

3) Decorator and Typehint. These are syntax for function's transformation. There are 2 considerations for these function's transformation.

a) Function transformations through decorators and typehints are usually applied as object properties or nested calls. Both cloudpickle and codepickle can handle these properties. However, in the case of codepickle, using 'inspect.getsource' still includes the source code with the decorator on top of the function. During exec, the decorator is executed again, leading to its application twice. A potential solution is to modify the source code by removing the decorator.

b) During serialization, there is no reference from a serialized function to the decorator class. This might result in a NameError during deserialization. The workaround is that the developer has to manually provide the reference to the decorator class along with the serialization process.

4) Objects without Name Reference, such as Lambda Functions.

We currently use exec to construct the function. Without a name reference, such as in the case of lambda functions, we are unable to retrieve the lambda function from the result and return it to the deserialization process. A potential solution is to use eval when a lambda function is detected.

We can establish a clear correlation between the experimental outcomes summarized in Table 1 and the previously identified limitations denoted as L.x.y.z, where “x”, “y”, and “z” specify point within that subsection. For example, L.1.a.i representing Limitation (1)(a)(i). This notation facilitates a more concise and reader-friendly presentation.

Out of 128 tests in the Cloudpickle test suite mentioned in III-B, there are 95 passed and 33 failed. The reasons for the failures can be mapped to the limitations identified earlier as follows.

Table 1: *Initial codepickle test result.*

Testsuite	Result
Function	pass
Function that calls another function	pass
Function that contains inner function	pass
Function that uses global variables	pass
Function recursive	pass
Class, user defined	pass
Imported modules	pass
Lambda function	L.1 and L.4
Function with decorator	L.3
Function with closure variables	L.2

1) Limitation of Getting Source Code from CPython (L1.a.1) - 3 cases: Notable cases include NamedTuple and dataclass.

2) Limitation of Getting Source Code from Func-

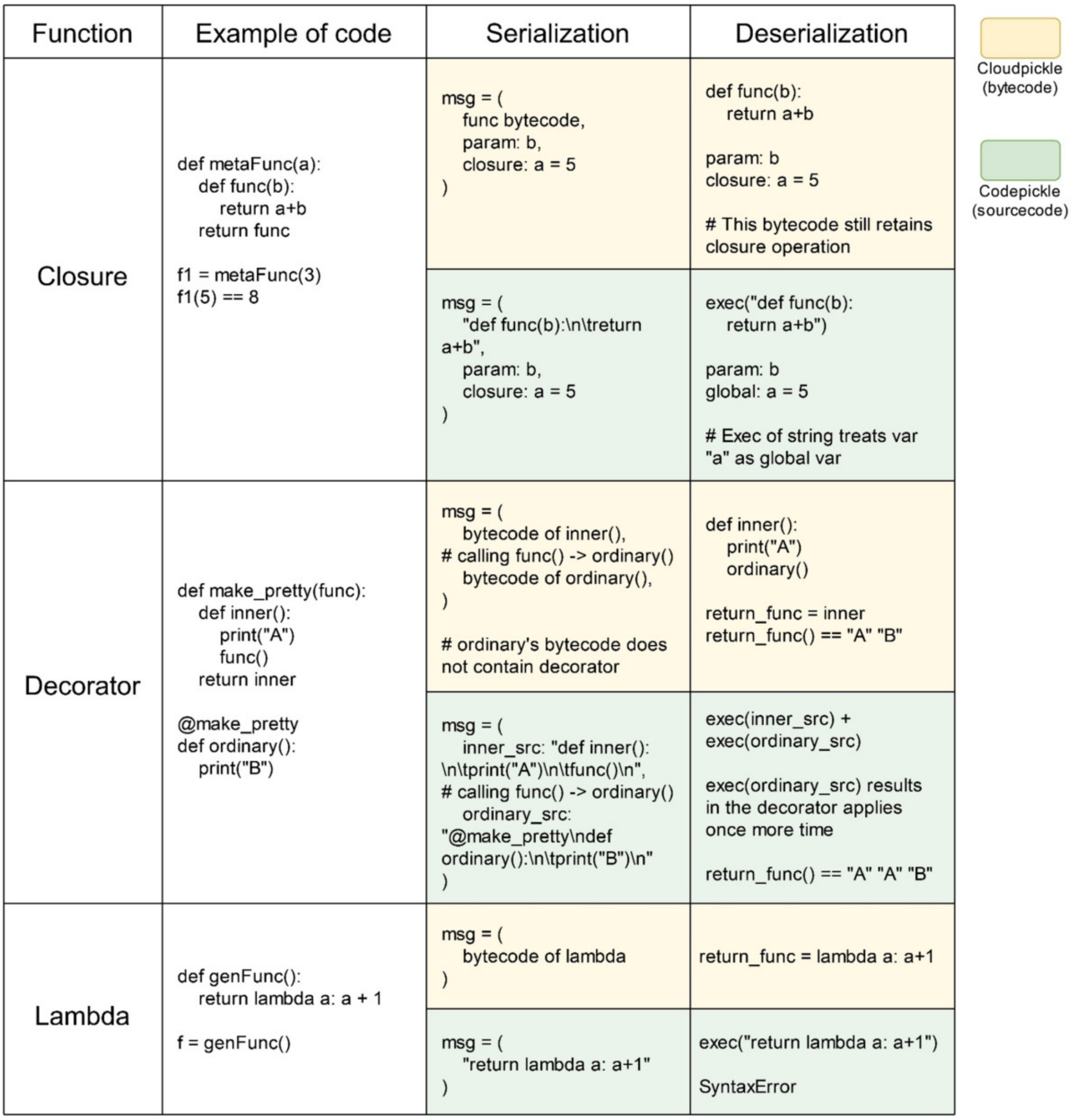


Fig.1: *Internal Process comparison between Cloudpickle and Codepickle.*

tion Constructed by exec or Codepickle Itself (L1.a.2) - 10 cases:

3) Limitation of Using Closure Variables (L2) - 3 cases: Notable case involves using super() for class instance initialization, which is internally a closure implementation. Other cases include side effects from changing closure to global variables.

4) Limitation of Decorators and Typehints (L3) - 9 cases: It is challenging to separate between L3.a and L3.b, so they are collected together. Notable limitations include the usage of decorators like '@staticmethod', '@property', and ABC abstraction, which are commonly used in Python development.

5) Limitation of Lambda Function (L1.b and L4) - 8 cases.

Intriguingly, the Cloudpickle test suite lacked any test cases specifically designed for nonlocal variables. We understand that 'nonlocal' syntax is indistinguishable from a closure variable in the bytecode format; however, it is different for the source code format. This highlights the importance of having an isolate test suite to prevent an overfitting implementation.

5. PERFORMANCE

We present a benchmark comparing the performance of our proposed approach, Codepickle, against several alternatives commonly used for handling function objects:

- Cloudpickle (standalone)
- Manual handling of source code
- Pyodide handling

Our primary focus areas are processing speed and message size. We use the tests similar to the kernel test suite mentioned in section III-A with few more test cases such as distributed sort which is mentioned as an example for distributed computing in Ray document[34]. We observe that the trends are almost the same for all function types for both speed and message size.

5.1 Processing Speed

From Fig.2 and Fig.3, the test results are consistent with the implementation of the Codepickle section, which has been further developed by combining functionality from both Cloudpickle and 'exec'. Fig.2 presents the time taken for 1,000 pickle processes of simple function and Fig.3 presents the time taken for 1,000 pickle processes of distributed sort function. The execution time of Codepickle is longer than both Cloudpickle and exec. For Pyodide, the 'exec' involves calling the AST function to modify the code beforehand, resulting in a longer execution time.

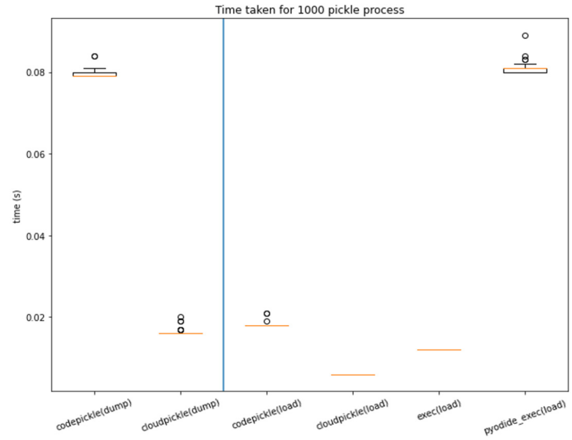


Fig.2: *Benchmark: Speed serializing simple function.*

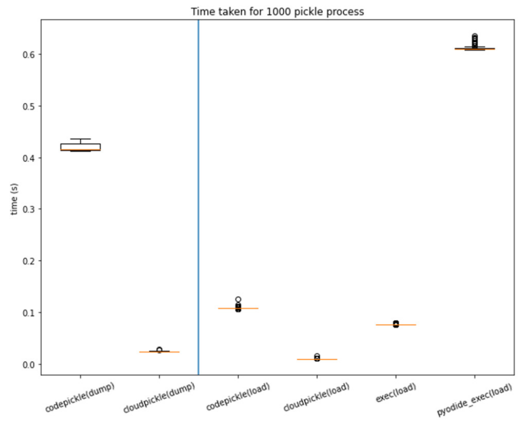


Fig.3: *Benchmark: Speed serializing distributed sort function.*

5.2 Message Size

From the Table 2, it was found that the size of the source code is smaller than the size of the code object data in general cases. The size of Codepickle message is around 84% of that of Cloudpickle for small messages. The notable exception is the distributed sort test case, where the source code size exceeds the code object size. This discrepancy is attributed to the large number of comments within the test case's function.

6. FURTHER IMPROVEMENT

Building upon the limitations identified in the previous section, we embarked on a systematic exploration of potential improvement strategies. To facilitate this investigation, we adopted a version control approach. Each proposed solution was implemented and tested within a dedicated Git branch. This modular approach enabled efficient experimentation and independent evaluation of each improvement. Subsequently, the promising modifications were integrated into the main codebase, culminating in the current iteration of Codepickle.

Table 2: *Benchmark: Size of pickled message.*

	Codepickle	Cloudpickle	Source code
Function			
Simple	470	533	34
With default argument	497	547	47
Chain functions	596	970	88
Functions with shared variable	627	723	111
Function Closure			
Closure	529	619	78
Function Recursive			
Self-recursive	520	620	73
Chain-recursives	689	810	157
Function with "import" module			
Simple	554	784	118
Using alias	561	793	120
Classes and instances			
Simple	662	715	60
Instance	665	718	NA
Function that uses class	949	1172	202
Inherited class	1199	1346	305
Instance of inherited class	1264	1392	NA
Selected benchmark			
Distributed sort	1512	1301	969

1) For L1.a.1, where a specific object type might require a custom dispatch method for serialization and deserialization, it's essential to explore the three main prerequisites:
   a) Can we differentiate this object from other objects? This involves determining if there is a distinctive characteristic or property that allows us to differentiate the specific object type from others. For example, examining the object's type using type(object).
   b) Can we retrieve all crucial information from an object and create the message? It is crucial to assess whether we can gather all the necessary information from the object to construct a comprehensive message for serialization.
   c) Do we have an object construction API or setstate that we can use to turn the message back into an object? To ensure successful deserialization, it's important to investigate whether there is an object construction API or a setstate method that can be employed to recreate the object from the serialized message.

   Unfortunately, after thorough investigation, it has been determined that both 'collections.namedtuple' and 'dataclasses' do not meet the prerequisites for implementing a proper dispatch method. As a result, addressing these issues is currently not feasible, and they will remain unresolved.
2) For L1.a.2, concerning function objects constructed using exec, there are two potential approaches to explore:
   a) Caching Source Code in Functions Created by exec in Codepickle: During the last deserialization process in codepickle, the source code can be saved as a cache into the object's properties. In subsequent serializations, if 'inspect.getsource' fails to retrieve the source code, the function can be inspected for a cached source code, which can then be used as an alternative. While this simple mitigation solves the exec issue originating from codepickle, it only addresses a subset of the problem. This approach reduces the number of failures from 33 to 27.
   b) Aggressive Approach - Overriding the exec Function: A more aggressive approach involves overriding the built-in exec function. Since it's not known in advance which objects will be used, the source code can be appended to any new objects created from exec. With this aggressive method, the number of failures is further reduced from 33 to 25. However, two failures persist:
      i) One case involves exec being done on a class. The source code is appended to the first-level object (the class), so the class's methods do not have the cache. We cannot recursively add the cache source code into those methods because the source code pertains to the class, not the methods.
      ii) Another case involves a test falling into the limitations related to Lambda (L1.b and L4).
3) For L3, addressing functions with decorators involves utilizing code modification. During function construction, the source code is inserted into the AST (Abstract Syntax Tree) module, and all decorators are subsequently removed from the first-level object. The choice of using AST instead of direct string/regex modification is made for accuracy. This approach results in a reduction from 33 fails to 28 fails.
4) For L1.b and L4 - Lambda, addressing the issues related to lambda functions is challenging. The source code obtained from 'inspect.getsource' for lambda functions is typically incomplete and cannot be accurately interpreted by the AST (Abstract Syntax Tree). While string manipulation is a possible solution, it is deemed out of scope for this research. As a result, the limitations associated with lambda functions remain unaltered.

Table 3: *codepickle: Python version compatibility test.*

		Serialize at version				
	Python version	3.7	3.8	3.9	3.10	3.11
Deserialize at version	3.7	96%	96%	98%	98%	98%
	3.8	96%	96%	98%	98%	98%
	3.9	96%	96%	98%	98%	98%
	3.10	89%	89%	90%	98%	98%
	3.11	87%	87%	89%	97%	98%

The consolidation of experimental improvement branches yielded a significant reduction in Cloudpickle test suite failures. The initial count of 33 failures dropped to 18, with the remaining issues primarily concentrated on lambda functions.

7. VERSION COMPATIBILITY ANALYSIS

This section explores the compatibility of Codepickle with different versions of pickles and Python environments.

Our primary focus is ensuring version compatibility, with the goal of verifying that Codepickle can successfully deserialize function objects serialized across different Python versions. Although we intended to assess Cloudpickle's compatibility as well, we encountered significant limitations. Attempts to manipulate incompatible bytecode resulted in unhandled segmentation faults. This poses a challenge for standard test frameworks, as segmentation faults cause the program to terminate abruptly, hindering data collection and halting subsequent unit tests. Consequently, we have deemed Cloudpickle testing beyond the scope of this study. In contrast, Codepickle's reliance on Python source code circumvents dependency on bytecode compatibility, where non-compatible code results in a Syntax Error rather than a critical failure. Utilizing a targeted test suite as mentioned in Section III-C, we successfully evaluated its portability across Python versions 3.7 to 3.11, as detailed in Table 3.

We can explain the results that Codepickle can almost fully support pickling functions across different Python version with few exceptions; as follows:

• The Pickle version used during serialization must be selected to ensure compatibility with the deserialization endpoint. For instance, Python3.7 supports Pickle version up to 3 while later version supports 4 and 5. Therefore, if an application is intended to deserialize data on a lower Python version, it must specify a compatible, lower Pickle version during serialization. Since our study did not focus on testing Pickle version compatibility, we standardized our tests to use Pickle version 3.

• Certain Python bytecodes and syntax are not supported in earlier Python versions. For example, Python 3.7 and 3.8 include an inplace-matmul bytecode that is not yet supported in earlier versions. Similarly, the "match" syntax was introduced in Python 3.9, so attempting to use it in versions prior to that will result in a syntax error. The same issue arises with the Exception Groups (Except*) syntax introduced in Python 3.11 .

• Additionally, there are other technical limitations previously discussed, such as handling closures and lambda functions.

8. CONCLUSION

Codepickle demonstrates promising potential for enhancing the efficiency and flexibility of Python in distributed computing environments, particularly within heterogeneous grids where different Python versions might coexist. Our evaluation across Python versions 3.7 to 3.11 showcased its successful portability. The ability to operate with source code offers a clear portability advantage over Cloudpickle. Furthermore, Codepickle's modular design facilitates potential future improvements and seamless integration with existing frameworks.

While Codepickle establishes a valuable foundation, further research is warranted to address limitations like closure support and explore advanced security considerations. Additionally, investigating the performance characteristics of Codepickle compared to Cloudpickle across diverse use cases would provide valuable insights for developers in the distributed computing domain.

AUTHOR CONTRIBUTIONS

Conceptualization, E.T. ; methodology, E.T.; software, E.T.; validation, E.T.; formal analysis, E.T.; investigation, E.T.; data curation, E.T.; writing—original draft preparation, E.T. and K.P.; writing—review and editing, E.T. and K.P.; visualization, E.T.; supervision, K.P. All authors have read and agreed to the published version of the manuscript.

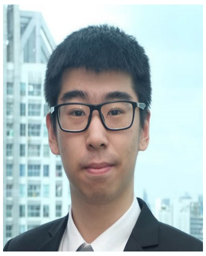
References

[1] A Géron, *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow*, O'Reilly Media, 2019.
[2] F. Nelli, *Python Data Analytics*, Apress, 2015.
[3] F. Chollet, *Deep Learning with Python*, Manning Publications, 2018.
[4] S. Raschka and V. Mirjalili, *Python Machine Learning*, Packt Publishing Ltd, 2019.
[5] J. VanderPlas, *Python for Data Science Handbook*, O'Reilly Media, Inc., 2016.
[6] P. Moritz *et al.*, "Ray: A Distributed Framework for Emerging AI Applications," *13th USENIX Symposium on Operating Systems Design and Implementation*, USENIX Association, pp. 561-577, Sep. 2018.
[7] M. M. McKerns, L. Strand, T. Sullivan, A. Fang, and M. A. G. Aivazis, "Building a Framework for Predictive Science," in *Proceedings of the 10th Python in Science Conference*, pp. 1-12, Feb. 2012.
[8] "PySpark Documentation — PySpark 3.5.1 documentation," Accessed: Apr. 25, 2024. [Online]. Available: `https://spark.apache.org/docs/latest/api/python/`
[9] "Using IPython for parallel computing — ipyparallel 8.9.0.dev documentation," Accessed: Apr. 25, 2024. [Online]. Available: `https://ipyparallel.readthedocs.io/en/latest/`
[10] "Joblib: running Python functions as pipeline jobs — joblib 1.4.0 documentation.," Accessed:

Apr. 25, 2024. [Online]. Available: https://joblib.readthedocs.io/en/latest/
[11] W. Stallings, *Data and Computer Communications*, Pearson Education, 2013.
[12] Y. Shafranovich, "Common Format and MIME Type for CommaSeparated Values (CSV) Files," *Request for Comments 4180*, RFC Editor, Oct. 2005.
[13] D. Crockford, *JSON: The Fat-Free Alternative to XML*, JSON.org, 2006.
[14] Extensible Markup Language (XML) 1.0 (Fifth Edition), W3C Recommendation, 26 November 2008. [Online]. Available: https://www.w3.org/TR/2008/REC-xml-20081126/
[15] cloudpickle. cloudpipe, 2023. Accessed: Apr. 25, 2024. [Online]. Available: https://github.com/cloudpipe/cloudpickle
[16] dill. The UQ Foundation, 2023. Accessed: Apr. 25, 2024. [Online]. Available: https://github.com/uqfoundation/dill
[17] R. Buyya and S. Venugopal, "A Gentle Introduction to Grid Computing and Technologies," *CSI Communications*, pp. 9-19, Jul. 2005.
[18] L. F. G. Sarmenta, "Volunteer Computing," Ph.D. dissertation, Massachusetts Institute of Technology, 2001.
[19] R. Schreiber, "Embarrassingly parallel processing: A new paradigm for massively parallel computers," *Citeseer*, 2003.
[20] A. Karlsson, "Embarrassingly parallel data analysis for transportation big data," *Transportation Research Part C: Emerging Technologies*, vol. 46, pp. 181–189, 2014.
[21] "Security - Ray 2.11.0," [Online]. Available: https://docs.ray.io/en/latest/raysecurity/index.html
[22] Anyscale team, "Update on Ray CVEs CVE-2023-6019, CVE2023-6020, CVE-2023-6021, CVE-2023-48022, CVE-2023-48023, Accessed: Apr. 25, 2024. [Online]. Available: https://www.anyscale.com/blog/update-on-ray-cves-cve-2023-6019-cve2023-6020-cve-2023-6021-cve-2023-48022-cve-2023-48023
[23] R. Abraham, "Running multiple projects with different python versions/ray versions and docker images," Accessed: Apr. 25, 2024. [Online]. Available: https://discuss.ray.io/t/running-multipleprojects-with-different-python-versions-ray-versions-and-docker-images/6253/3
[24] K. Asanović *et al.*, "The Landscape of Parallel Computing Research: A View from Berkeley," techreport UCB/EECS-2006-183, Dec. 2006. [Online]. Available: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-183.html
[25] "Pyodide is a Python distribution for the browser and Node.js based on WebAssembly," Accessed: Apr. 25, 2024. [Online]. Available: https://github.com/pyodide/pyodide
[26] E. Tangmunchittham and K. Piromsopa, "An Analysis of Python Serialization towards Distributed Systems," *2022 19th International Conference on Electrical Engineering/Electronics, Computer, Telecommunications and Information Technology (ECTI-CON)*, Prachuap Khiri Khan, Thailand, pp. 1-4, 2022.
[27] "The Python programming language," Accessed: Apr. 25, 2024. [Online]. Available:https://github.com/python/cpython
[28] Oracle. (n.d.), Java Object Serialization, Accessed: Apr. 25, 2024. [Online]. Available: https://docs.oracle.com/javase/8/docs/platform/serialization/spec/serialTOC.html
[29] PHP Manual, (n.d.), Serialization, Accessed: Apr. 25, 2024. [Online]. Available: https://www.php.net/manual/en/language.oop5.serialization.php
[30] G. V. Rossum and F. L. Drake, *The Python Reference Manual*, Centrum voor Wiskunde en Informatica Amsterdam, The Netherlands, 1995.
[31] "Python Functions," Accessed: Aug. 28, 2023. [Online]. Available: https://www.w3schools.com/python/python_functions.asp
[32] "Python Functions - GeeksforGeeks," Accessed: Aug. Aug. 28, 2023. [Online]. Available: https://www.geeksforgeeks.org/python-functions/
[33] "Python version numbers and their encoding ("magic number")," Accessed: Apr. 25, 2024. [Online]. Available: https://github.com/google/pycnite/blob/main/pycnite/magic.py
[34] "Pattern: Using nested tasks to achieve nested parallelism – Ray 2.34.0," Accessed: Aug. 11, 2024. [Online]. Available: https://docs.ray.io/en/latest/ray-core/patterns/nested-tasks.html

APPENDIX A

- https://github.com/Bankde/codepickle - This repository serves as the primary development environment for the Codepickle module. It encompasses the kernel test suite, benchmark framework, bytecode analysis across various major Python versions, and the portability test suite.
- https://github.com/Bankde/cloudpickle - We have cloned the Cloudpickle repository and applied the necessary patches to integrate Codepickle into the code. Please refer to the related branch for more details on the implementations.



Eakasit Tangmunchittham received his Bachelor's degree in Computer Engineering from Chulalongkorn University in 2016. Following graduation, he began his career as a computer engineer, focusing on distributed database development using C language. He later transitioned to the role of a penetration tester, specializing in web application security. Eakasit also served as a public speaker for OWASP Thailand, where he presented on the topic of "Race Conditions." In 2019, he pursued a Master's degree in Computer Engineering at Chulalongkorn University.

Krerk Piromsopa received the BEng and MEng degrees in computer engineering from Chulalongkorn University, Thailand, in 1998 and 2000, respectively. In 2003, he was awarded a scholarship from the Royal Thai Government for pursuing his PhD degree in computer science, which he received from Michigan State University in 2006. He has been a lecturer and later a professor at Chulalongkorn University since 2001. His research interests are in computer security, computer architecture, distributed systems, and big-data in health-care applications. He has been a member of the IEEE and the IEEE Computer Society for 20 years.