# Handling Software Icebergs

Jaak Henno [1], Hannu Jaakkola [2] and Jukka Mäkelä [3]

[1] *Tallinn University of Technology, Ehitajate tee 5, 19086 Tallinn, Estonia*
[2] *Tampere University, Pori Campus, P.O. Box 300, FI-28101 Pori, Finland*
[3] *University of Lapland, Rovaniemi, Finland*

**Abstract**

Once upon a time programming was done just writing commands of a programming language in a proper order, but currently software is created using libraries, API-s (Application Programming Interface), frameworks, Dockers, Kubernetes etc. Libraries load other libraries, API-s call other API-s and as a result seemingly short and simple programs may have amazing depth of code and complexity, what causes for programmers many problems, especially for students. For interpreted code this depth could be (approximately) measured with the ratio of the visual code vs code in libraries. It is shown that the number of LOC (Lines Of Code) in invisible code – code in libraries, modules, API-s etc. is even in small practical programs thousands-millions times greater than the number of lines in the visible code. Innovations (cloud computing, multicore CPU-s etc.) cause introduction of new libraries and modules which enable use of new possibilities in existing software ecosystem, but also introduce bigger and bigger amounts of invisible code. During the pandemic grow student's use of WWW tutorials, but abundance of code in programming examples/tutorials on WWW is sometimes unnecessary, caused by obsolete or unneeded packages and libraries and sometimes also by desire to earn on adverting (un-needed, but popular) code packages for high-paying customers. In the following are analyzed some Python 3 and JavaScript examples.

**Keywords**

Software complexity, libraries, modules, API, LOC

## 1. Introduction

With exhaustion of Earth's natural resources humanity is constantly increasing scientific research, which nowadays is based on world-wide programming and communication ecosystem. Once upon a time programming was done just writing commands of a programming language in a proper order. Modern software is created using libraries, API-s (Application Programming Interface), selecting commands from menus in frameworks, packed with Dockers, Kubernetes etc. Libraries load other libraries, API-s call other API-s and as a result programs have amazing depth of code. For not-compiled (interpreted code) that depth could be (approximately) measured with the ratio of the visual code vs code in libraries - the total number of lines used in visible code is thousands-millions times greater than the number of lines in the visible code. This abundance of code is often unnecessary, caused by obsolete or unneeded packages and libraries, but sometimes also by desire to earn on adverting (un-needed, but popular) code packages for high-paying customers. This creates many difficulties for students, who often cannot understand the programming task but try to solve it using 'top-down' programming – first import all libraries and packages, what have been used in some previous project and then just start googling trying to get from Internet sources as many snippets (which may be useful) as possible. Such 'top-down' programmers often do never become a 'real'

*SQAMIA 2022: Workshop on Software Quality, Analysis, Monitoring, Improvement, and Applications, September 11--14, 2022, Novi Sad, Serbia*
Emails: [1]jaak.henno@taltech.ee (corresponding author); [2]hannu.jaakkola@iki.fi; [3]jumakela20@gmail.com
ORCID: 0000-0003-0188-7507 (Hannu Jaakkola)

programmers, but continue creating new packages and libraries, which 'just work' and further contaminate the software ecosystem.

## 1.1. World-wide ecosystem of programming and communication

With exhaustion of Earth's natural resources humanity is constantly increasing scientific research. Spending on sciences has increased in last four years worldwide by 19% and the number of scientists grew by 13.7% reaching currently already to 8.8 million [1].

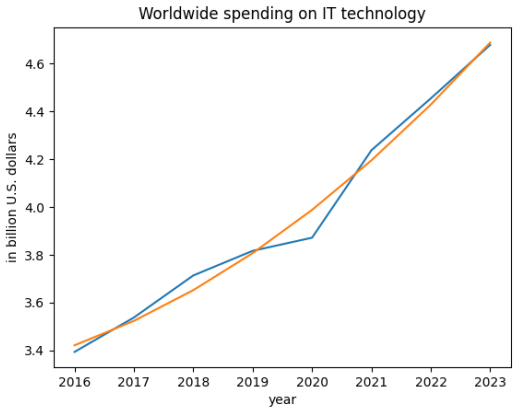


Fig. 1. Wordwide spending on IT technology is rapidly growing (the yellow line – linear approximation, growth > 18%))

Modern Science is based on worldwide ecosystem of programs and communication and spending on IT technology is growing rapidly. Since the start of 2020, the COVID-19 crisis further boosted this trend. We constantly need more and more programmers. But software development today is orders of magnitude more complex than one or two decades ago and is increasing with every new innovation in hardware, new method in software, new encryption method.

## 2. Changes in programs and in programming

Once upon a time programming was done just writing commands of a programming language in a proper order, but these times are long gone. While e.g. the language Python 3.7 has only 35 keywords and 59 symbols, the real power of the language is in its libraries and modules, e.g. in the computer used to write this text are installed 361 Python 3.7 modules. Currently programming is done with modules, libraries, API-s, Dockers, Kubernetes etc. Every module presents some functionality what is (usually) far more complex than a single command of a programming language. Modules and API-s call other modules, libraries, API-s, what makes modern programs complex hierarchical systems, where the visible part (code what is inspect able in the program) is a minuscule part of the invisible in final code, hidden in libraries, API-s etc. Modern software is like icebergs in Antarctic – the main part is invisible.
In the following are presented some (simple) examples showing this 'iceberg-style' programming and an analyse of the reasons for programs 'blow-up', since the depth/complexity is causing many problems for students, but also for the whole modern world-wide software ecosystem. To illustrate the issue here is introduced program's visibility index: ratio of visible code vs code loaded in libraries/modules.

## 2.1. A simple graph

The following graph depicts development of the SQAMIA conference over 2012..2019 in terms of number of pages in the conference proceedings; the yellow line is a second-order (square) approximation of the process.

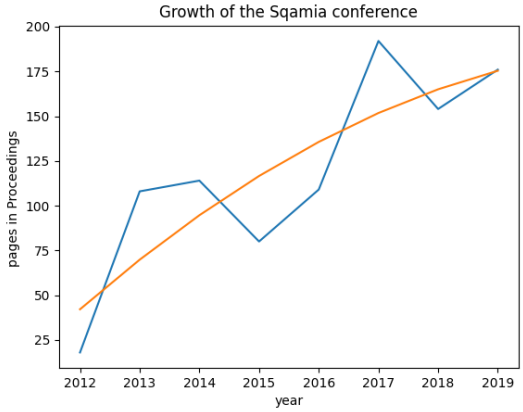


Fig. 2. Growth of the SQAMIA conference –by number of pages in Proceedings (the yellow line – second-order approximation)

The above graph was produced by 17 lines (including commentaries – they are important to make a program easier to understand for humans) of Python 3 code :

```
 1.   import numpy as np
 2.   import matplotlib.pyplot as plt
 3.   # x axis values
 4.   x_sqamia = range(2012,2020)
 5.   # corresponding y axis values
 6.   y_sqamia = [18,108,114,80,109,192,154,176]   # Sqamia
 7.   plt.plot(x_sqamia, y_sqamia)
 8.   # naming the x axis
 9.   plt.xlabel('year')
10.   # naming the y axis
11.   plt.ylabel('pages in Proceedings ')
12.   # giving to graph a title
13.   plt.title("Growth of the Sqamia conference")
14.   # calculating coefficients of the approximation line
15.   a2, a1, a0 = np.polyfit(x_sqamia, y_sqamia, 2)
16.   plt.plot(x_sqamia, a2*x_sqamia*x_sqamia + a1*x_sqamia + a0)
17.   plt.show()
```

This text is understandable also for non-programmers. But programmers looking the above code will wonder – no loops ??? The 'classical' program, i.e. using only commands from a programming language without any libraries would have to use several loops – using the `x_sqamia` iterator, projecting values to axes, solving the system of equations for creating square approximation in order to calculate correlation parameters `a2,a1,a0`. Totally is hidden the AI (Artificial Intelligence) – calculating the size and visually pleasing placement of axes and their scales.

All the functionality of the above program is hidden in the two explicitly called libraries `numpy`, `matplotlib` and many others, called from these two. The `pyinstaller` [2] tool reveals, that for drawing of this graph are 162 modules called all together:

`bootlocale` • `_collections_abc` • `_weakrefset` • `abc` • `codecs` • `collections` • `collections.abc` • `copyreg` • `encodings` • `encodings.aliases` • `encodings.ascii` … … `sre_constants` • `sre_parse` • `tokenize` • `traceback` • `types` • `warnings` • `weakref` …

Tracing the files for these modules revealed, that they all together contain 18378 lines of Python code (together with commentaries) and the added modules increased 31882 times the size (on disk) of the program – from 2 KB to 63764 KB.

The 'blow-up' and then 'missing' together make programming (at least for students) a non-deterministic exercise, thus difficult/impossible to learn) is in (this case mainly) caused by modern exceedingly diverse executing environments – Windows (32/64 bit), MacOS, GNU/Linux, AIX, Solaris, FreeBSD and OpenBSD, IBM-style mainframes, diverse lot of mobile devices, which all have some differences, thus for them are called different modules. Many of them are specific to current environment, e.g. `ntpath` exists only on Windows, but the `posixpath` exists only on

Posix/Unix/Mac systems and is missing in Windows environment (but the compiler is still trying to load it). This creates (in every environment) many modules which compiler marks as 'missing'. There are several types of missing modules:

- optional - imported within a try-except-statement
- delayed - imported from within a function
- conditional - imported within an if-statement
- top-level - imported at the top-level; they are most important and may cause the program misfunction

The same module may fall into several categories, e.g. for this program the missing module `org.python` was called by module `copy` (optional), `xml.sax` (delayed, conditional) and by module `setuptools.sandbox` (conditional).
In Windows 64bit environment was the `PyInstaller` able to access only 41 packages (ca 25%). The `PyInstaller` (in the same environment) even searched for more and did not find 238 modules, but it also warned "`This does not necessarily mean this module is required for running you program`" – missing modules are required in some other operating environment, e.g. the list of modules contained 125 encodings:

`encodings.ascii • encodings.base64_codec • encodings.big5 • encodings.big5hkscs • encodings.bz2_codec • encodings.charmap • encodings.cp037 • encodings.cp1006 • encodings.cp1026 …`

(missing were codecs for Egypt, Cuba, UAR etc, ).
All together were among missing modules 150 optional, 106 delayed, 94 conditional and 313 top-level. Large number of missing top-level modules does not mean, that the code can not be executed – a missing top-level module were called from a module, which itself is optional.
But the highest level of program's execution is the OS (Operating System, often called also as Operating Environments) [3], in this case Windows 10 64bit. Microsoft also uses several options in different OS-es. The program's manifest shows, that Microsoft approves the code in 5 Operating Systems (all Windows versions before Windows 7 are already missing) :

```
<supportedOS Id="{e2011457-1546-43c5-a5fe-008deee3d3f0}"/> <!--  Windows Vista -
->
<supportedOS Id="{35138b9a-5d96-4fbd-8e2d-a2440225f93a}"/><!--  Windows 7 -->
<supportedOS Id="{4a2f28e3-53b9-4441-ba9c-d69d4a4a6e38}"/><!--  Windows 8 and
Windows Server 2012 -->
<supportedOS Id="{1f676c76-80e1-4239-95bb-83d0f6d0da78}"/><!--  Windows 8.1 and
Windows Server 2012 R2 -->
<supportedOS Id="{8e0f7a12-bfb3-4fe8-b9a5-48fd50a15a9a}"/><!--  Windows 10,
Windows 11, Windows Server 2016, Windows Server 2019 and Windows Server 2022 -->
```

In modules found by `PyInstaller` where all together 18378 LOC (Lines Of Code), thus the index of code visibility, i.e. relation to what in code is visible and what is hidden is rather astonishing:

$$visibility = \frac{LOC(visible\_code)}{LOC(invisible\_code)} = \frac{18}{18378} = 0.0000979431$$

It is rather difficult (e.g. for students) to understand fully a program when they can inspect only 0.0000979431-th size part of it – e.g. what should be done with the program for the above graph in order to include two missing years 2020, 2021 to it ? It turns out, that a quite new concept – data filtering – is needed, and from inspecting the above program (as an educational simple example for producing data graphs) this is not easy to produce.

## 2.2. A simple web page

Unnecessary lines of code may be also inserted deliberately – either by ignorance of authors or by desire to advertise some popular library.
A typical program for beginners (using any programming language/system) is to produce a program to create a visible display "`Hello World`". In html5 for this are needed only some lines of html5 code (in basic mode).

But the programming powerhouse `Facebook` (currently `Meta)` needs thousands of such files (with largely increased functionality for tracking users). Thus they created a Javascript+Html5 library `react.js` advertising it with "*React makes it painless to create interactive User Interfaces.*" [4].
The www-tutorials site W3Schools introduces `react.js` with an example to create just this example for visible display of string `"Hello World"` [5] using 17 lines of html5 code.

```
<!DOCTYPE html>
<html>
  <head>
    <script src="https://unpkg.com/react@18/umd/react.development.js"
crossorigin></script>
    <script src="https://unpkg.com/react-dom@18/umd/react-dom.development.js"
crossorigin></script>
    <script src="https://unpkg.com/@babel/standalone/babel.min.js"></script>
  </head>
  <body>
    <div id="mydiv"></div>
    <script type="text/babel">
      function Hello() {
        return <h1>Hello World!</h1>;
      }
      ReactDOM.render(<Hello />, document.getElementById('mydiv'))
    </script>
  </body>
</html>
```

This script has some strange features:

- (lines 11:.13) - why for displaying a static string "Hello World!" (without any variables) is created a function in an environment `ReactDOM` – the html5 language (without any Javascript) can perfectly show a static string in a pre-defined div?
- (line 14) - why is for displaying/rendering the string used a proprietary function "`ReactDim.render`" – in an html-document rendering anything on screen is always done by browser, a script can only use the available CSS functions (CSS is not used)?
- (lines 4..6) - why for creating this functionality are loaded three Javascript libraries with altogether 73571+3358+26275 = 103204 lines (after un-minifying) of JavaScript code, which are not needed at all?
- (lines 4..5) - the `crossorigin` scripting has been for years considered extremely unsafe [6], so why it us used in a worldwide tutorials site proposed/presented to many students?
- the example is missing the `title` tag, which by the html5 standard is obligatory (missing `</title>` should cause the browser to ignore the rest of the page [7])

The (absolutely excessive, hidden) lines of Javascript code make this example extremely unhealthy:

$$visibility = \frac{LOC(visible\_code)}{LOC(invisible\_code)} = \frac{17}{103204} = 0.000164722$$

Using plain html5 would return an easily understandable script, producing exactly the same page:

```
<!DOCTYPE html>
<html>
  <head>
    <title> Hello !</title>
  </head>
  <body>
    <div id="mydiv">Hello World!</div>
  </body>
</html>
```

$$visibility = \frac{LOC(visible\_code)}{LOC(invisible\_code)} = \frac{11}{11} = 1.0$$

The main burden – 73571 lines of code (after un-minifying) comes from the Babel library for interpreting Ecmascript ES6 (the official name for Javascript). All major browsers can handle ES6 already for last five years [8], thus Babel is not needed at all [9], it only increases page's size and slows it down.
At least for a student this is not "painless", but (very) painful.

## 2.3. Increase of complexity with AI

Our computing environments are constantly changing and their complexity is rapidly growing. Every day are uploaded/modified thousands of new libraries/API-s which are also used in new programs, e.g. in one week - August 3 – August 10, 2022 in the Github repository for node.js "*1,479 files have changed and there have been 34,550 additions and 6,640 deletions.*" [10].

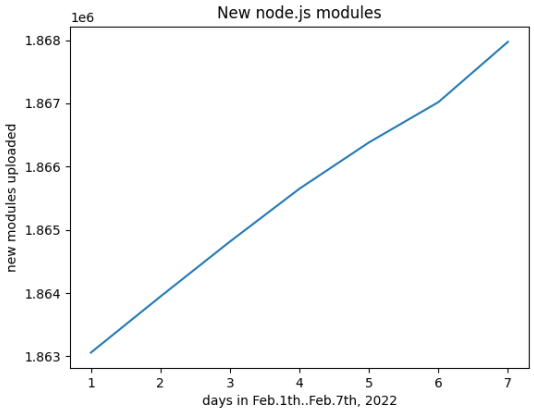


Fig. 3. Uploading of new node.js modules during Feb.1..Feb.7, 2022 [11]

With exhaustion of Earth's natural resources humanity is forced to use their resources/technologies far more efficiently. All our innovations begin with communication – a workgroup meeting, chat in wine/bierstudbe etc, i.e they are presented in some natural human language. For handling human language in computers have been developed programs with enormous (inner) complexity.
The Tensorflow example from Google [12] tries 'understand' natural language' - any language – English, Serbian, Croatian etc without using any dictionaries; these 'superlinguists' abilities are based on 4 imported modules only:

```
import tensorflow as tf
import numpy as np
import os
```

The `PyInstaller` reveals that from these four are imported other 158 modules:

```
_bootlocale • _collections_abc • _weakrefset • abc • codecs • collections •
collections.abc • copyreg • encodings • encodings.aliases • encodings.ascii •
encodings.base64_codec • encodings.big5 • encodings.big5hkscs •
encodings.bz2_codec • encodings.charmap • encodings.cp037 • encodings.cp1006 •
encodings.cp1026 • encodings.cp1125 …•
```

The module `os` imports also package `boto` (the current version is `boto3/botocore`), which appears in the modulegraph gross-reference list 334 times. The module `boto` is the low-level CLI (Command Language Interface) for several cloud platforms - AWS (Amazon Web Services), i.e. SDK (System Developing Kit), which makes it easy to integrate a Python library or script with AWS services including Amazon S3, Amazon EC2, Amazon DynamoDB, but also for Google Storage and some private cloud systems, e.g. VMware vCloud, OpenStack, Open Nebula or

Eucalyptus. Thus Tensorflow is already prepared to swim in Amazon cloud (boto is a small river dolphin living in Amazon [13]).
The search for Python 3.7 modules was able to find 55 of 158 modules with 24465 lines of code. The Tensorflow example program had 52 lines of code, thus in this case

$$visibility = \frac{LOC(visible\_code)}{LOC(invisible\_code)} = \frac{52}{24465} = 0.002125485$$

But here the main growth in complexity comes from the program.
Definition of the Tensorflow model contains only 6 lines of code:

```
def build_model(vocab_size, embedding_dim, rnn_units, batch_size):
model = tf.keras.Sequential([tf.keras.layers.Embedding(vocab_size,
embedding_dim,batch_input_shape=[batch_size, None]),
tf.keras.layers.GRU(rnn_units,return_sequences=True,stateful=True,recurrent_init
ializer='glorot_uniform'),tf.keras.layers.Dense(vocab_size)]
return model
```

But this definition creates a model with over 4 million parameters as shown in output for Python command `model.summary()`:

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding (Embedding)        (128, None, 256)          43776
_________________________________________________________________
gru (GRU)                    (128, None, 1024)         3938304
_________________________________________________________________
dense (Dense)                (128, None, 171)          175275
=================================================================
Total params: 4,157,355
Trainable params: 4,157,355
Non-trainable params: 0
```

Handling more than 4 million parameters is (very) slow in 'plain' Python, since Python uses only a single CPU core and cannot parallelize computation. In most of modern computers are already multicore CPU-s and their speed can be utilized with another library - bodo, which is [14]: "*a new compute engine using a novel JIT inferential compiler technology that brings supercomputing-like performance and scalability to native Python analytics code. Bodo automatically parallelizes Python/Pandas code allowing applications to scale to 10,000+ cores and petabytes of data.*"
Innovations – cloud computing, multicore CPU-s, use of graphics etc introduce constantly new modules and libraries – and hide more LOC 'under the water'. The rapidly growing cloud computing is currently one of the most important reasons increasing complexity of programs introducing new libraries. The recent forecast [15] predicts, that the global cloud computing market will be 1,251.09 billion USD by 2028 or the annual growth over the period will be 19.1%. The growth of cloud computing in education is expected even bigger – 25.6% [16]. In 2021 nearly half (42%) of enterprises in EU already used cloud computing [17].

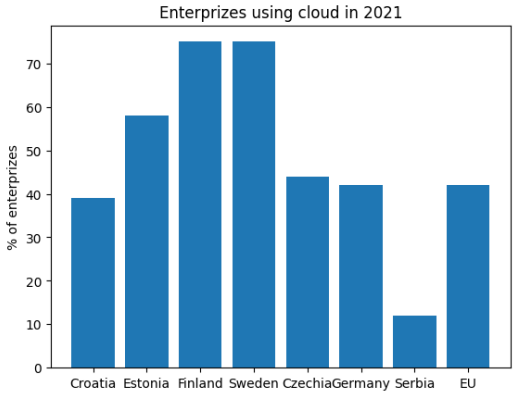


Fig. 4. Percentage of enterprizes using cloud computing

## 3. Conclusions

Many (most) of the classical programming textbooks considered programming as a process of creating a linear artefact - putting programming language statements one-after-another in a proper order.
But already creators of the compilers for the first high-level language – Fortran, Cobol – understood that programs and their functionality (also creating the programs itself) are far from linearity and programs are always presented as trees – e.g. as the AST (Abstract Syntax Tree).
Programming, human thought has never been linear, although many of created by ourselves tools (standards, texts, constitutions etc.) are trying to force to us this style of thinking.
We are not linear. Programming has constantly becoming more and more non-linear - hierarchical. Programmers still have to create linear text, but programs are including libraries, IDE-s, modules etc. which all make programs functionality (inner working) strongly multidimensional. Software development is not a smooth or simple process, many researchers are worried with its growing complexity, see e.g. [18]. Emily Freeman, head of DevOps at Amazon Web Services Inc characterized it "It's a study in entropy, and it is not getting any more simple"[19]. Unfortunately we are still not aware how to teach to our non-linearly thinking students the proper non-linear thinking methods/styles, but they should (at least) be aware of tremendous dimensionality of our programs (and our thoughts).

## References


[1] Unesco Institute for Statistics. 2022. How much does your country invest in R&D. Retrieved from http://uis.unesco.org/apps/visualisations/research-and-development-spending/ on July 30th, 2022.

[2] Pyinstaller. 2022. Pyinstaller 5.2. Retrieved from https://pypi.org/project/pyinstaller/ on July 30th, 2022.

[3] Microsoft. 2022. Application Manifest. Retrieved from https://docs.microsoft.com/en-us/windows/win32/sbscs/application-manifests on July 30th, 2022.

[4] React. 2022. React - A JavaScript library for building user interfaces. Retrieved from https://reactjs.org/ on July 30th, 2022.

[5] W3Schools. React Getting Started. Retrieved from https://www.w3schools.com/react/react_getstarted.asp on July 30th, 2022.

[6] rot, Rémy. 2020. Understanding Cross-Origin Resource Sharing Vulnerabilities. Tenable Blog. Retrieved from https://www.tenable.com/blog/understanding-cross-origin-resource-sharing-vulnerabilities on July 30th, 2022.

[7] MDN Web Docs. 2022. <title>: The Document Title element. Retrieved from https://developer.mozilla.org/en-US/docs/Web/HTML/Element/title on July 30th, 2022.

[8] W3Schools. 2022. JavaScript Versions. Browser Support for ECMAScript 2016. Retrieved from https://www.w3schools.com/js/js_versions.asp on July 30th, 2022.

[9] SidmartinBio. 2022. Do you need Babel for ES6? Retrieved from https://www.sidmartinbio.org/do-you-need-babel-for-es6/ on July 30th, 2022.

[10] Nodejs. 2022. Nodejs/Node Pulse. Retrieved from https://github.com/nodejs/node/pulse on July 30th, 2022.

[11] Node. 2022. Node.js® is a JavaScript runtime built on Chrome's V8 JavaScript engine. Download for Windows (X64). Retrieved from https://nodejs.org/en/ on July 30th, 2022.

[12] TensorFlow. 2022.Text generation with an RNN. Retrieved from https://www.tensorflow.org/text/tutorials/text_generation on July 30th, 2022.

[13] Wikipedia. 2022. Amazon river dolphin. Retrieved from https://en.wikipedia.org/wiki/Amazon_river_dolphin on July 30th, 2022.

[14] Pypi. 2022. Bodo 2022.6 Project Description. Retrieved from https://pypi.org/project/bodo/ on July 30th, 2022.

[15] Research and Markets. 2022. Cloud Computing Market Size, Share & Trends Analysis Report by Service (SaaS, IaaS), by Enterprise Size (Large Enterprises, SMEs), by End Use (BFSI,

Manufacturing), by Deployment, and Segment Forecasts, 2021-2028. Retrieved from https://www.researchandmarkets.com/reports/5397840/cloud-computing-market-size-share-and-trends on July 30th, 2022.

[16] Markets and Markets. 2022. Cloud Computing in Education Market by Service Model (SaaS, PaaS, and IaaS), Deployment Model (Private Cloud, Public Cloud, Hybrid Cloud, and Community Cloud), User Type (K-12 and Higher Education) and Region - Global Forecast to 2021. Retrieved from https://www.marketsandmarkets.com/Market-Reports/cloud-computing-education-market-17863862.html on July 30th, 2022.

[17] Eurostat. 20232. Cloud computing used by 42% of enterprises. Retrieved from https://ec.europa.eu/eurostat/web/products-eurostat-news/-/ddn-20211209-2 on July 30th, 2022.

[18] Carey, Scott. 2021. Complexity is killing software developers. Infoworld. Rerieved from https://www.infoworld.com/article/3639050/complexity-is-killing-software-developers.html on July 30th, 2022.

[19] Amy-Vogt, Betsy. 2021. 'DevOps for Dummies' author Emily Freeman introduces revolutionary model for modern software development. siliconAngle. Retrieved from https://siliconangle.com/2021/09/29/devops-dummies-author-emily-freeman-introduces-revolutionary-model-modern-software-development-awsq3/ on July 30th, 2022.