

COLLECTING PROGRAMMING DIAGNOSTICS FOR STUDENTS LEARNING TO
PROGRAM: A DISSERTATION SUBMITTED IN PARTIAL FULFILMENT OF THE
REQUIREMENTS FOR THE DEGREE OF BSc (HONS) COMPUTER SCIENCE

by

Lawrence Thorpe
THO14585438

The purpose of this project is to collect information about programming errors and code changes made in response to those errors, through an extension for *Visual Studio*; collecting 'big data' about common errors faced by students in their programming projects.

School of Computer Science



UNIVERSITY OF
LINCOLN

University of Lincoln
2019

Contents

1	Acknowledgements	3
2	Abstract	4
3	Introduction	5
4	Literature Review	6
5	Methodology	9
5.1	Project Management	9
5.2	Software Development	9
5.3	Toolsets and Machine Environments	10
5.4	Research Methods	13
6	Design, Development and Evaluation	16
6.1	Requirements elicitation, gathering, collection and analysis	16
6.2	Design	16
6.3	Development process and implementation details	18
6.4	Evaluation	20
7	Project Conclusion	22
8	Reflective Analysis	23
9	References	25
A	Appendices	27

1. Acknowledgements

I would like to thank Phil Carlisle for suggesting the topic which forms the basis for the project, and for supporting development of the project and particularly the artefact, through continuous feedback and suggestions. I would also like to thank Mingjun Zhong, my supervisor, for supporting much of the academic parts of the project, and helping to make sure I was on track with the writing as well as the development aspects of the project.

2. Abstract

The problem of indeterminate, inexplicable or difficult to understand errors is something that is often faced by those new to programming. This project aimed to collect data about programming code and errors generated by this code, in order that it could be used as 'big data' which could be used for machine learning. The focus of this project was primarily on the collection of this data; through the use of a tool—an 'extension'—for collecting errors in the software *Visual Studio*, which is commonly used by those learning to code, and, in particular, in education.

The extension collects code and error data automatically when a build takes place. This data is stored in the cloud for its ability to scale dynamically as data grows; additionally, data is stored in a way that allows for tracking changes made to files and corresponding errors over time. Throughout testing, the *Visual Studio* extension was able to collect data which was used for statistical analyses.

Utilising the data collected by this project, it could be possible, for example, to link commonly occurring errors with additional context that would otherwise not be present with just those generated by *Visual Studio*; including those errors which can be considered difficult to decipher from compiler output alone. Following this, it could be conceivable to use this data to expand further, to provide automatic suggestions to fix these errors based on the data collected from many users and what common resolutions other people come to.

3. Introduction

Many people in the process of learning how to program have difficulty in understanding the errors in their code; often because the generated errors are hard to interpret or require additional context. In many cases, learners use the software *Visual Studio* IDE ('Integrated Development Environment') to develop their code. In *Visual Studio*, it is possible to program in a variety of languages; with examples including C++ and C# which are supported by default. Often, using just errors generated by the compiler when the code is built, it can be difficult to establish what went wrong and how to go about solving the issue; and, even worse, in some cases, the errors can be unrelated to the code itself, instead based on contextual information—such as the flow of code—which is not accurately reported in the error itself; for example, a missing bracket or semicolon; these issues are similarly discussed in the paper by Campbell et al. (2014).

One of the main motivations for this project is that the situations around which so-called 'unrelated' errors form are generally not very well understood; this is because a 'simple' compiler does not take into account the context of the code. Instead, a simple syntax mistake—such as a missing bracket or semicolon—can spawn a series of unrelated errors—often-times throwing off novice, but even well-versed programmers—a jarring focus shift from the code actually being developed—particularly if trying to focus on and learn the fundamentals of programming. Many times, fixing these kinds of errors involves working out which particular syntax causes it with little guidance from the errors themselves, although, as can be imagined, this is not very efficient or practical, and can be quite time-consuming. For this reason, the project aims to collect data which can be helpful in clarifying some of the uncertainty in which these errors commonly occur. By collecting 'big data' about code and errors, it could be possible to, for example, train a machine learning model to understand the context in which errors can occur— a possible use case for this might be to suggest potential fixes for resolving errors, based on changes made by people in the past.

The project's artefact consists of a few key components; namely: the *Visual Studio* extension, which collects the errors; cloud storage, which is used to store the errors and code themselves; and finally, a website, which demonstrates some of the ways the data could be used— in this case, through some basic statistical analysis, similar to that discussed in the paper by (Jadud, 2005).

The goal of the client, Phil Carlisle, is to gather data from *Visual Studio* to be used in future projects, in order to develop better understanding of programming errors. In accordance with the requirements of the client, many of the project aims consist primarily of ensuring that the client's requirements are met satisfactorily. Namely, these requirements include ensuring that the artefact is able to collect data as intended, and, that its operation can generally be considered automatic; that is to say, with little to no further input from the end user— making it suitable for use in, for example, a lab/learning environment, as opposed to the dedicated 'test' environments often used for collecting similar data. As well as this, the structure of the data is a key requirement, focussing on which attributes of the errors and code are collected, as well as how they are tracked over time— the aspect of time being used in a similar way to the 'snapshots' discussed in the paper by Ahadi et al. (2015).

Another aim was the development of a series of software components which together form the basis for actually collecting the data. This aim was broken into a number of objectives, focussed around the development process itself, including: provisioning the cloud resources used to store the data collected—importantly, considering the ability to scale with demand and storage requirements; finding a suitable compression algorithm (and software library) for storing the full source code; sourcing a software library for finding differences in code; and, most importantly, development of the extension itself, linking in with the aforementioned components. Finally, the objective of the website (<https://lncn.ac/codeanalysis>) is to provide some insight into how the data collected might be used; providing a way to visualise the data in the form of some statistics.

4. Literature Review

This section discusses the research done around the topics covered in development of the artefact. The project covers a range of fields, with such topics ranging from collecting code errors; storing the collected data, including using cloud storage techniques; management of big data; and, the use of compression techniques. An example of the common workflow used to link these items within the artefact is as follows: collect errors via *Visual Studio* error output as well as program code; compress this data; before uploading to the cloud storage—such storage must be considered based on its ability to easily scale to the size of data uploaded to it—i.e. allowing for big data.

Research on the collection of data pertaining to code errors is varied; for example Jadud (2005) collects code errors using the 'BlueJ' tool. Although the paper mentions some of the details collected by the tool, Jadud makes no mention of the specificities of the methodology used. The tool, 'BlueJ', collects data including but not limited to; complete source code, computer user name, compilation indexing (an incrementing number for each time the program is compiled), compilation result (success/failure), timestamps, IP address, and line numbers of errors. This list (ibid., 28) provides some insight into the pieces of data that can be used to create a similar tool, and helps to understand what makes a group of data unique to a user, while avoiding as far as possible storing personal information, such as an alternative method for tracking users like a real name or email address.

This project aims to expand on the previously mentioned studies, and collect similar data to that which has been collected before, but with the addition of collecting changes made by students to fix errors. While some research has been done around the collection and statistical analysis of code errors (Jadud, 2005; Becker, 2016b), previous studies do not have as much of a focus on the aspect of collecting 'big data' and using such data to directly feed back new information and/or assist users. These studies do, however, tend to give more information about the statistics gathered from their datasets, providing some insight into how the data can be used, beneficial to understanding what needs to be stored and how data can be structured.

Further, in a different paper, Becker discusses common errors faced by students (2016a); the paper has a specific focus on 'novice' programmers so provides some insight into which errors may be encountered by those developing early programming skills while learning. The paper also summarises some of the error types found, as well as a collation of the reasons many repeating errors may be caused.

Expanding on the research done by Jadud and Becker, Ahadi et al. (2015) builds on the concept of 'source code snapshots' proposed by Jadud, and particularly the use of them to determining the performance of students. The paper applies various machine learning techniques to the data to find with high accuracy students that are not performing as well as hoped, and thus may benefit from additional assistance.

Further to this, Castro-Wunsch et al. (2017) go on to use neural networks to classify data in a similar subject, i.e. predicting students that may be likely to require assistance. The paper finds that the algorithms used were successful in identifying with relatively high accuracy students that were likely to need assistance with programming based on existing data. Although not strictly related to the goals of this project, the paper's findings prove that it is possible to apply machine learning techniques to data of this nature.

The paper by Campbell et al. (2014) provides some insight into the concept of making error messages more useful than those simply generated by the compiler; a potential future use of the data that is to be collected by the project. For example, Campbell et al. draws attention to the fact that many compiler messages are unclear in their meaning or are difficult to interpret, and are often given for lines that are not directly relevant to the actual error, such as where a symbol (e.g. '{', '}' or ';') is missing. The paper uses a language model to predict where certain tokens may be expected—for example, the signature of a function is almost always expected to be followed by an opening '{'—as opposed to parsing the code itself.

This project differs from the previously mentioned papers in that it additionally includes the aspect of collecting the *changes* made by users; with an intent to track how changes to the code affects errors generated— an example being that adding a semicolon might remove an error or even a multitude of errors, for instance. Importantly, by adding the element of time, it is possible to discover from the data how code is changed over time, and which changes to code can be correlated to which errors being added, or in fact removed. The artefact differs from other previously mentioned works in that it directly interacts with *Visual Studio*—for students both using it for assignment work as well as personal projects, which might be used for experimentation—as opposed to, for example, a dedicated test environment. Similarly, the artefact is designed to be language agnostic; that is, it should be capable of collecting errors from any programming language that *Visual Studio* is capable of generating errors for, regardless of syntax or types of error. Data is considered ‘raw’ with little to no semantic meaning such as type of error, as that can be inferred later during the data mining process. As another goal of the project is to collect data specifically for the purposes of ‘data mining’, the storage of such data is an important consideration, and as such, both the format and method for storing data is an important consideration; specifically, which features of the data are collected and which are ignored— this is because the data can be transformed as required.

There is a wide range of research surrounding the topic of data storage and, in particular, cloud storage. Hashem et al. (2015) discuss using cloud for storing ‘big data’— of particular note this paper states that cloud storage can be of most use when there is difficulty maintaining sufficient storage locally; this is relevant as the procedure to establish sufficient storage would likely be problematic, especially considering the relatively short time-frame of the project.

Contrastingly, Hashem et al. mention the higher cost potential (being billable) and possible security challenges due to outsourcing resources to a cloud provider (2015, 107), however ensuring that the resources are managed effectively with little to no external input means this is a likely worthwhile trade-off for this project. Further to this, the project scope is short term and resources are subject to change. For this reason, it is important that storage resources be provisioned quickly and with little upfront cost.

The cloud was a good fit for this project due to the aforementioned reasons; this is backed up by Pawar et al. (2016, 2), stating that the user only pays for what is required, without first investing a large sum; which would be impractical for this project as such resources may not be required after the short time-frame. In their paper, Tak et al. discuss the concept of underutilisation and, more specifically, over-provisioning—having more computational power than required—which is particularly prevalent in small workloads or businesses (2013, 1228). By using the cloud therefore, over-provisioning can be avoided for the relatively small workload of the project; though if demand were to increase, the cloud would be capable of expanding to the new demand, as will be discussed in the following section.

When considering how the data was stored, the maintainability and management was considered, which, for such time and resource constraints as this, is particularly relevant. Muñoz-Escóí and Bernabéu-Aubán (2017) discuss ‘PaaS’ or ‘Platform as a Service’, the concept of providing a tool or resource— particularly a cloud resource—like a database, as a service, with all that which a service entails; that is to say, it is managed by the provider. Such services can be automated to include dynamic or ‘elastic’ scaling of resources to demand (ibid., 622).

When storing large amounts of (‘big’) data, it is generally considered important to consider the consequences of storing such quantity of data; as Bhat (2018, 55) states, the rate at which magnetic storage capacity increases annually is becoming a problem as data requirements increase. Although Bhat (2018, 59) mentions the problems of compression when done within data centres—including that while size is reduced, and therefore decreases disk I/O, it consequently increases CPU load for compression/decompression for the data centre—these problems are somewhat mitigated by performing such compression/-decompression on the user’s computer *before* the data is uploaded or *after* downloading.

To compress the data generated by the artefact—which consists of mostly text (code)—one of a variety of text compression algorithms could be used. Shanmugasundaram and Lourdasamy (2011) compare some of the common algorithms used for text compression; investigating the resultant output sizes of algorithms with different types of data. Such algorithms are considered ‘lossless’, i.e. algorithms which produce a result from which the original data can be reconstructed, with no loss. The paper (ibid., 68-72) details some of the compression techniques used by the algorithms for compressing data.

Research around the topic of *Visual Studio* errors is relatively sparse, although the composition of such errors can be considered relatively simple, consisting of only a few components— which could be considered one of the leading reasons for difficult in understanding them; as information contained within the errors themselves is limited. Microsoft (2017b) describes the physical data structure for errors themselves accessible through their extension programming interface, allowing some understanding of

how they are structured as well as with which data types and how they are stored. Similarly, Microsoft (2015) provides a list of errors in the *C#* programming language, giving a brief description of each—allowing understanding these errors somewhat better in context. On the other hand, however, relatively simple representation of the errors themselves means that they can be considered independent of each other, and this makes these errors relatively easy to represent as a simple data structure consisting of only a few fields. For this reason, code is collected alongside the errors themselves, in order to provide context; the line number of each error is known, with the code can on that line (and surrounding lines) available for context; combined, this information could be used to provide more useful heuristics than a simple error alone, when a large number of samples (‘big data’) are collated.

5. Methodology

5.1 Project Management

This section talks about the aspects of the project which were managed, and, further, it discusses how the project was managed. Finally, some detail is given on the principles applied in development of the artefact.

To maintain the status of the project and ensure that it was kept on track, a project plan was created consisting partly of week-by-week objectives (Thorpe, 2018, 4). The process for managing the project consisted of keeping track of each objective's process by breaking up each objective into a series of sub-tasks which are more manageable– e.g. over a few hours or a day. For tracking these tasks within those objectives, the project management process often involved the use of *Trello*, a project management tool which is easy to access from multiple devices, and also allows for note-taking and settings dates for tasks to be completed (Trello, 2019).

The project was managed using the kanban system. This system allowed visualising how different tasks progressed, and in which state each task was, for example; *To Do*, *In Progress*, *Waiting*, or *Done* (Nyström, 2011, 11). One disadvantage of using kanban as a project management tool is that of the overhead of breaking up the objectives into tasks. In cases where an objective is relatively simple but consists of many steps, it could be argued that the overhead of breaking up the objective into many small tasks, and writing each down–possibly including dating them–could introduce more overhead than the task itself is worth.

Using a kanban board for the project management allowed for managing many aspects of the project; this is because it can easily be applied to almost any set of tasks which could be represented by, as an example described by Nyström (2011, 11), sticky notes– this includes both the software development and management of the additional tasks that needed to be completed, including meetings to discuss setting up resources such as cloud services.

As the project is primarily a software development item, almost all aspects of the process could benefit from project management. A good example of how the project management process was applied, is that most of the aims and objectives were set through the requirements from the client, with the overall goal being to complete all of these tasks for the project to be considered 'complete', with client feedback (discussed further later) influencing the flow and change the direction of the project as it progressed.

One of the software engineering principles applied in the project management process involved the use of the iterative process; this meant that the program is not worked on and released as one monolithic artefact; but, instead, as Fowler and Scott (2000) states, improved over time, corresponding to feedback. More detail about the development process and the application of this process will be discussed in a later section.

Particularly key in the development of the artefact was the principle of separation of concerns; that is to say, for example, breaking components into smaller parts, getting such parts to a 'working' state, before refining and improving code– for example, to improve performance or quality of code. This was especially relevant for a team consisting of one member, as, as Miller (1956) states, a person is able to store around seven plus or minus two symbols in their working memory; so breaking up components into smaller parts and working on those first helped to maintain focus and integrity, without the need for too much context switching.

5.2 Software Development

This section discusses the software development process, and namely, describes the processes used to develop the software, fitting with the aspects of existing software development methodologies.

The artefact creation took place using an agile approach to software development. One of the reasons for choosing this particular approach is due to the limited time allotted to the project; as it can be more difficult to spend time on thoroughly following steps, such as the long-term planning of another approach such as the waterfall method. The agile process corresponds well with the iterative process mentioned in the previous section. Nyström (2011).

Agile processes fit well with the management aspects provided by the kanban system, which as Polk (2011) describes, can be simplified as a more complex project board, able to be added to almost any existing project using agile. In general, however, the concept of breaking objectives into tasks means that it is significantly easier to use them in the planning of agile sprints—i.e. deciding which tasks were to be done within each sprint (which could be simplified as how many, and which tasks are completed per week).

In order to gather requirements for the software—and to formulate that into a set of features to work towards—early on in the process of software development, relatively regular meetings (usually weekly) were set up with Philip Carlisle (the ‘client’)—these could be considered similar to the stand-up meetings of agile as referred to by Sutherland et al. (2007). When it was agreed that the understanding of the direction of the project was well defined and had become more-or-less ‘fixed’, the regularity of meetings decreased—with future meets focussed primarily on the feedback process, as discussed later.

In planning the project, an overview for how the artefact would be designed and how the data would be structured was created, although, this plan included provisions for the fact that requirements might have been likely to change during the process. By anticipating potential changes, it is possible, as Buckley et al. (2005, 315-316) state, to accommodate these foreseen changes in the design of the software when planning takes place and the objectives are set. While there was a set of requirements for the artefact, they are likely not considered strict enough to follow a plan-based methodology, and in particular, it makes sense to use an agile approach for its ability to adapt to changes easily and quickly.

The requirements gathered from the client for the project focussed primarily on how the data was intended to be used—therefore it was always a priority to collect the necessary data, with a focus on how the aforementioned data was structured; although, as mentioned later, the collected data can be transformed to work with in different ways. Other requirements were based around how the extension should operate—for example, a key requirement was that the data should be collected automatically; that is to say, without user input. Another requirement was to ensure that the extension is language-agnostic, meaning it works with any language supported by *Visual Studio*.

A particular aspect of agile that is important is that the client is able to frequently see changes made in each release, as well as to receive feedback. In a process similar to that mentioned by Khan et al. (2012) in describing the agile XP process, product demonstrations were created reasonably regularly, to show the current feature set and design, both to the project supervisor and the client. This process aided significantly in confirming that the artefact development was on track with the plan and deadline, and also to ensure the features required by the client were as expected.

The key deliverable item for the client and ultimately for end-users, is a packaged version of the *Visual Studio* extension which is installable with no further input; an installed copy of the extension is able to collect data immediately upon its successful installation.

The artefact development process itself did not require any specific computer hardware other than a machine capable of development work, namely one able to run *Visual Studio* and its experimental instance alongside it (used for debugging the extension which is installed in the experimental instance). Aside from the development machine, cloud resources were required to store the data collected by the extension.

Finally, regarding end-user devices; the only significant requirements to run the artefact include: the ability to run *Visual Studio* (although it might be safe to assume that students will already have this installed if they are learning to program, or that it will be installed in a lab environment); the ability to install extensions into *Visual Studio* (by either the end-user themselves or by administrators, for example in a lab environment); and, that there is a constant connection to the internet (also in most cases this can be assumed true—especially if for example on a university campus) in order to access the cloud.

5.3 Toolsets and Machine Environments

This section will discuss the tools used during the software development process—this includes for example the programming languages and environments, as well as some of the other services used within the development of the project, such as cloud services. Further, it discusses tools used during the project

Figure 5.1: Example of cloud bucket names generated from project GUIDs

- ☐ codeanalysis-14585438-1347217e-a24c-4af4-9f8f-a17965d6dc44
- ☐ codeanalysis-14585438-2dc44132-ff97-4343-a4c4-a3e58eee610d
- ☐ codeanalysis-14585438-5c17786c-7b12-4212-be4f-ce4cdfbad485

management process, as well as drawing comparisons on some of the possible tools that could have been used as alternatives.

As the artefact is an extension to the *Visual Studio* IDE (an ‘Integrated Development Environment’ widely used for programming), *Visual Studio* is also used to develop the extension, primarily as it provides native debugging support and development for extensions written for it. For similar reasons, *C#* is the suggested programming language for writing extensions for *Visual Studio* (Microsoft, 2017a; Microsoft, 2017c); this is ideal as it integrates well with the IDE and provides a lot of flexibility. A possible drawback for using *C#* is that it is JIT (‘Just-In-Time’) compiled—meaning that, as Troelsen (2005, 11-13) states, code is transformed into *C#*’s ‘Common Intermediate Language’ which is interpreted by the machine on which the program runs (although of note this has the benefit of being platform-agnostic)—as compared to a language requiring compiling to native machine code like *C++* (Hamilton, 2003, 21). This can mean a slight decrease in performance, while still remaining faster than an entirely interpreted language such as *Python*; this is unlikely to be problematic in this use case, however, as the overhead of *Visual Studio* itself is likely closely matched.

As there is little documentation covering the process of developing an extension for *Visual Studio*, an important part of the development process was developing an understanding of how various components fitted together, and how to perform the required functionality; for instance how to link code with events triggered by *Visual Studio*, namely the ‘on build’ and ‘after build’ events.

For the purposes of cloud storage of the data, *Google Cloud Storage* buckets were used, which allow for storing the files within a project as data ‘objects’, with the ability to store such objects in a structured way, similar to the layout of a file system. Other considerations for storing cloud data could be *Amazon S3* (Amazon, undated) or *Microsoft Azure Blob Storage* (Microsoft, undated[a]); however in the case of this project, *Google Cloud* services were chosen as access to these services was available at no extra cost through the university, whereas using an alternative would involve additional costs.

Each project created within *Visual Studio* has its own unique project GUID (‘Globally Unique Identifier’) (Microsoft, undated[b]) that identifies a project distinctly from other projects—something that simply a name could not guarantee, as multiple people might use that same project name, even multiple times themselves—which is used as part of the name for bucket to uniquely identify the files of one project from another. An example of some bucket names generated from project GUIDs are shown in Figure 5.1.

While the *Visual Studio* extension directly communicates to the cloud storage, for convenience, an API (‘Application Programming Interface’, commonly used to provide a set of well defined methods to allow communication between applications, which allows web-based access to the data from the cloud storage) was created utilising *Google Cloud Functions* (Google Cloud, 2019b). One major motivation for using *Cloud Functions* was that authentication with other *Google Cloud* services is automatic, provided the function is within the same cloud project. Another reason for using the service is that such functions are relatively easy to write and deploy using the *Node.js* environment (with *JavaScript* as the language). The design, as well as further detail on the implementation of this API, will be discussed in further detail later.

To compress the code data, a lossless text compression algorithm was used—this is because it must be possible to obtain the original version (Shanmugasundaram and Lourdasamy, 2011) (i.e. without any changes), while maintaining small storage requirements for potentially large code files to keep costs down and avoid storing more than is necessary (Bhat, 2018). The *BZip2* compression library was chosen for this project primarily due to its high compression ratios. To test various algorithms, the file `fields.c` (Appendix A.1) from the Canterbury Corpus (Bell, 1997) was used— the contents of this file is not important for the purpose of this project (in fact it is not relevant to the project itself), other than the fact that it is a code listing; however, of note is that data from this corpus is specifically designed for the use of testing lossless compression algorithms. The results of testing these algorithms for compressed size and speed of compression are shown in Table 5.1. Each of the algorithms were

Table 5.1: Comparison of speed and size of various lossless compression algorithms, each algorithm was run at level 9 (highest compression)

Algorithm	Time (seconds)	Compressed size (bytes)
Uncompressed	–	11,150
Zip	0.008	3,275
GZip	0.006	3,136
BZip2	0.130	3,039

Table 5.2: Comparison of *Git* hosting providers

	GitLab	Bitbucket	GitHub
Price for private repos (per month)	Free	Free	\$7
Cloud hosted	Yes	Yes	Yes
Maximum repository size (GB)	10	1	100
Per file size limit (MB)	None	None	100

run at their highest compression levels (i.e. level 9), as that meets this particular project's needs, as previously mentioned; compression to the smallest size, regardless of speed. It is worth mentioning that the algorithms listed are limited to those supported by the *SharpZipLib* C# library (Kreuger, undated)—this means that other algorithms such as *LZMA* are not included due to the limited library support provided for such algorithms in C#. The reasons for using the *SharpZipLib* library are: the library is open-source MIT licensed, meaning it can be included in a closed source project such as this; the library is available from the *NuGet* package manager, allowing for easy install; and, finally, that it closely resembles the syntax of existing C# functionality, such as the built-in functionality of `System.IO.Compression`.

For project management and handling code, the *Git* version control system was used—allowing for tracking changes to code over time. As Spinellis (2012) discusses, there are many advantages to using *Git* for version control, including, for example, relevant to this project of one developer: being able to see history of changes; perform different tasks in branches which can be later combined; and, manage, edit or reorder changes. Spinellis also mentions the advantages of using a hosting service to upload the *Git* repository as a secure backup. The *Git* repository was hosted in the *GitLab* cloud environment—this was chosen in particular due to the fact that it supports free hosting for private repositories (GitLab, undated); this was an important consideration as the project is considered closed-source (as opposed to a free, open-source project), meaning code is protected and private as expected by the client. Alternative hosting providers include *Bitbucket* and *GitHub*, however at the project's inception, *GitHub* private repositories involved a cost (although this has since changed). Similarly, *Bitbucket* provides up to 5 users with free private repository hosting—however, on the other hand, *GitLab* is provided free for an unlimited number of users. *GitLab* was also chosen for this project due to the additional storage space provided over, for example, *Bitbucket*—the main reason for this was due to the large number of files required, including libraries and builds of the extension, which were hosted in the repository. A comparison of the three primary *Git* hosting providers considered for this project are listed in Table 5.2.

As mentioned previously, *Trello* was used to organise the project management process. Some of the reasons for choosing *Trello* for this particular project include: it works on multiple devices, as it is hosted in the cloud and provides mobile apps; it allows for a visual representation of the state of each task with its progress, and where applicable a due date; and, that it is free to use for the features required within the scope of this project. Johnson (2017) discusses some of the shortcomings of using *Trello*, particularly for large projects—namely the fact that management of tasks can become cumbersome as the size of a project increases—although as the project was somewhat limited in scope by time, these shortcomings were not as impactful. Another point that Johnson brings up is that there is no way to mark a task 'complete' which is mainly of concern to projects involving many people; but as the project development took place with one person only, the need to share the specificities of the workflow is negated.

A comparison of various tools that could be applied to this particular project and a comparison of some of the features relevant to the project are listed in Table 5.3.

Table 5.3: Comparison of project management tools

	Trello	Jira	Microsoft Project	Pivotal Tracker
Primary functionality	Kanban	Issue tracking	Gantt chart	Agile stories
Lowest price (per month)	Free	\$10	£5.30	Free
Cloud hosted	Yes	Yes	Yes	Yes
Mobile apps available	Yes	Yes	No	Yes
Due dates	Yes	Yes	Yes	Yes
Ability to mark task completed	No	Yes	Yes	Yes

5.4 Research Methods

The following section discusses the methods used within the project for the aspect of collecting data, and in particular it discusses the types of data used, and how these datum are collected and used. Further, some descriptions of how the data might be used are included, along with some example figures created which show some basic statistics, to show some of the data collected so far.

One of the aims involved creating a tool to be used in the collection of data from a range of users, with the aim of collecting the data from around 1,000 users of the *Visual Studio* extension. Further to this, as previously mentioned, as documentation on the process of creating a *Visual Studio* extension was limited, a key part of the process involved collating information from various—sometimes unofficial—sources in order to extract information with which to develop the components of the artefact as set out in the requirements. This process, as can be imagined, involved a lot of research around various sources of information; gathered knowledge, further, could not generally considered complete enough to be used alone—this meant that it was necessary to undertake additional steps to develop and understanding of how to combine the various components of that comprised the final artefact together. Despite this, however, due to time constraints in the project, some compromises had to be made in the development of the artefact that resulted in slightly less than desired code quality—this will be discussed in greater detail in a later section.

Other research carried out included gathering information on how *Visual Studio* formats, stores, and collates/outputs its errors. This research was both important to understand how to integrate the extension with *Visual Studio*, as well as to better understand the workings of errors to understand how, and under which situations, they are generated. This understanding provided some insight into how data was formatted and stored, as well as understanding for how it might be processed in the future, and, finally, how it could be used to generate meaningful statistics.

The artefact collects quantitative data; that is to say, data collected by the *Visual Studio* extension is simply text data from the user's code. This is as opposed to qualitative data which would be considered subjective – an example of this might be how 'good' the code is, which is extremely difficult if not impossible to measure. As such, data will also be considered objective—i.e. is uploaded as 'raw' and unprocessed—with no further subjective input.

The data is largely 'nominal' (existing in name only, i.e. no value); namely, errors and code snapshots. It is possible, however, to transform the data in a few ways to make it more useful, and gather information from it—in fact this is likely to be the primary way the data is used, as on its own it is mainly useful for reconstructing a Git-like view of code and errors over time within a project. An example of how the data might be transformed is to aggregate all errors together (per project or globally), select the error codes, to discover how many occurrences of each error code exist (Figure 5.2); this would become therefore a form of ratio data, where there is zero or more of each error code, the occurrences of each code can be directly compared. Figure 5.2 was created by collecting the errors for every file in every project, performing some restructuring of the data to collect all such errors into one list, and then by counting and plotting the unique error codes provided by *Visual Studio*. More information on how these figures were created is provided later.

A further example of how data could be used is by plotting the number of errors over time (Figure 5.3), which could for instance show the decrease in errors as a user improves. Although Figure 5.3 shows the number of errors for all projects—which gives a general overview of how the extension is used—a more useful plot when analysing the data on a larger scale might be to limit the plot to one specific project GUID, to understand the trend in number of errors over time.

The transformed data will likely be used with machine learning, for example by analysing the data

Figure 5.2: Example of error code counts for multiple projects (C# & C++)
Occurrences of error codes in all projects

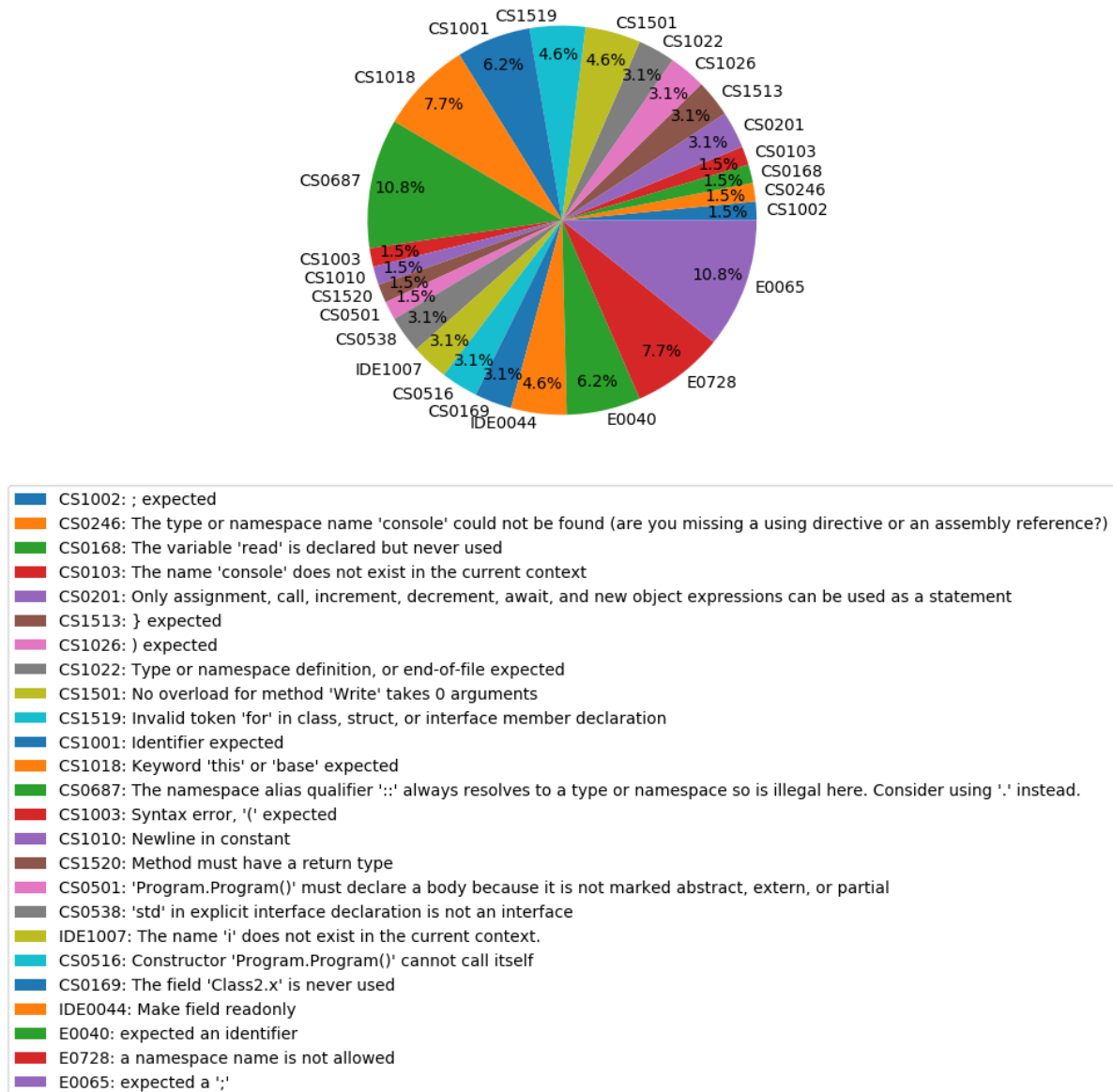
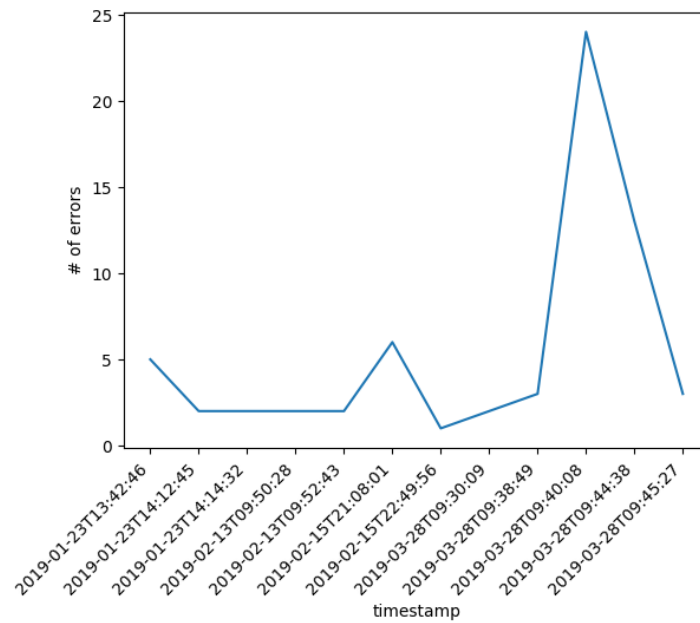


Figure 5.3: A plot of the total number of errors in all projects over time
Number of errors over time



of many users it might be possible to train a model of common errors based around certain types of syntax. The trained model could then be extended to provide suggestions that could be made to code in real-time, based on similar changes people have made in the past.

6. Design, Development and Evaluation

In this section, an overview of the design process, including gathering the requirements for the project's artefact, as well as the process of development and software components used to create such artefact, will be covered. This section also discusses how the extension for *Visual Studio* will be integrated into the existing layout of the IDE. Following this, the evaluation will take place in the form of a description of the testing that has taken place, and some initial, rough analysis of the data collected by the artefact.

6.1 Requirements elicitation, gathering, collection and analysis

Many of the requirements were put forward by Philip Carlisle (the 'client'), who initially proposed the software idea for the Project. Requirements were partially oriented on which data should be collected by the extension, so that it can be made usable in the future for analysis. Further gathering of requirements was performed through asking questions of what was needed; especially focussing on the data required to be collected by the client and how the extension was expected to operate. Further requirements were gathered continuously throughout the artefact development via an agile process—primarily through showing demos of iterations in the development the artefact itself, and basing new requirements on client feedback.

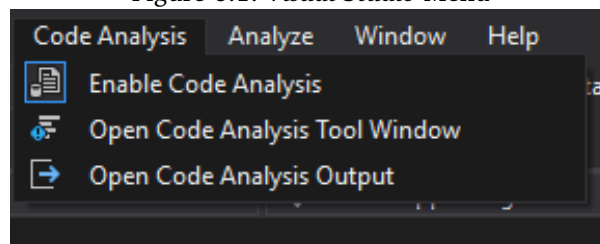
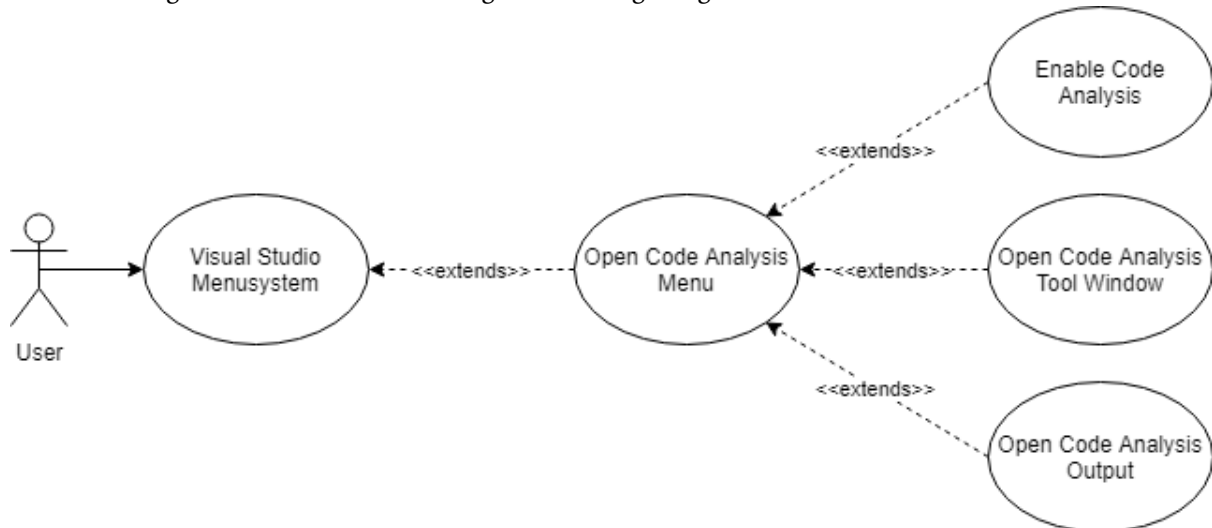
An initial plan was created as a rough guide for how the data could be structured, whereas the design for other elements was intended to be similar to the design of existing *Visual Studio* elements to maintain familiarity—for this reason the design was not as thoroughly planned before-hand, because the design for these elements was changed according to the limitations of the implementation which were discovered throughout the development process.

In the analysis of requirements; each requirement was translated into actionable objectives—a good example of this is the process for converting the data requirements into a programmatic representation of the data (in this case, in the form of a class). The plan's objectives were split into individual tasks which were easier to work with, especially in the agile workflow. This process within agile involved breaking down into individual development tasks, as described by Sutherland et al. (2007). An example of was in splitting the task of implementing the menu system into: researching the process involved; and then implementing menu items each as separate development tasks—in this case, it meant that each of these tasks was considered as being enough work for a small number of story points (Raza et al., 2017), or, put another way, a reasonable amount of time (usually, a few hours or under a day of work).

6.2 Design

The design of the extension itself (the artefact) was aimed to match the existing style of *Visual Studio* to help with familiarity for users that have experience with *Visual Studio* already. The extension was designed to be integrated with existing menus within the IDE (Figure 6.1) for quick access to the various functionalities of the extension, in a way that is both familiar and easy to use for existing *Visual Studio* users.

The way the extension integrates into *Visual Studio* is by extending an additional menu to the existing menu system in the *Visual Studio* (Figure 6.2), which contains 3 menu items. The first; '*Enable Code Analysis*', is the primary toggle of the features of the extension; this option allows a user to either enable or disable the functionality of the extension—that is to say, if such functionality is disabled, the extension will not collect any data. Secondly; the '*Open Code Analysis Tool Window*' provides access to one of the

Figure 6.1: *Visual Studio* MenuFigure 6.2: UML use case diagram showing integration with *Visual Studio* menus

primary features of the *Code Analysis* extension itself. The contents of this tool window are mentioned in more detail later but, in summary, the window is used to view the data generated by the extension in a user friendly way. The third menu item; '*Open Code Analysis Output*' is simply a shortcut item, allowing the user to open the existing *Output* window within *Visual Studio*, and automatically selects the output type as only the output from the *Code Analysis* extension itself.

Additionally, an output window item (Figure 6.3) was added both to record events as they happen, and for the additional benefit of aiding in debugging. This window is shared with other events in the editor such as build tasks, so it benefits from sharing an area with other similar information that can all be seen together.

Finally, the extension (artefact) contains a 'Tool Window' (Figure 6.4) which could be considered similar in function to a simple 'Git' viewer. The tool window allows a user to view the code and errors over time for all of the projects contained within their currently opened *Visual Studio* solution. For an individual file, the window will show the code itself (left pane), along with the corresponding errors for that point in time (right pane), as well as any patches (top tabs)—the code of the file with code changes applied incrementally up to a certain point in time—if applicable, and the corresponding errors for the code at that point in time.

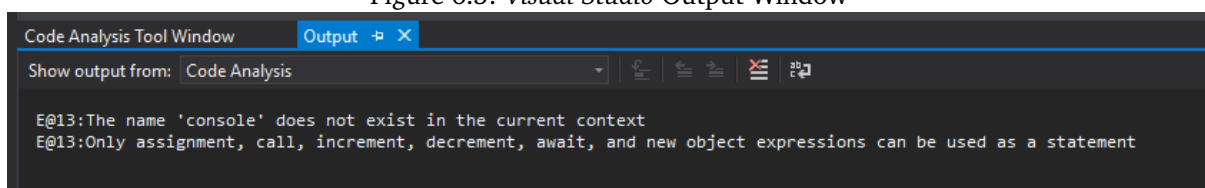
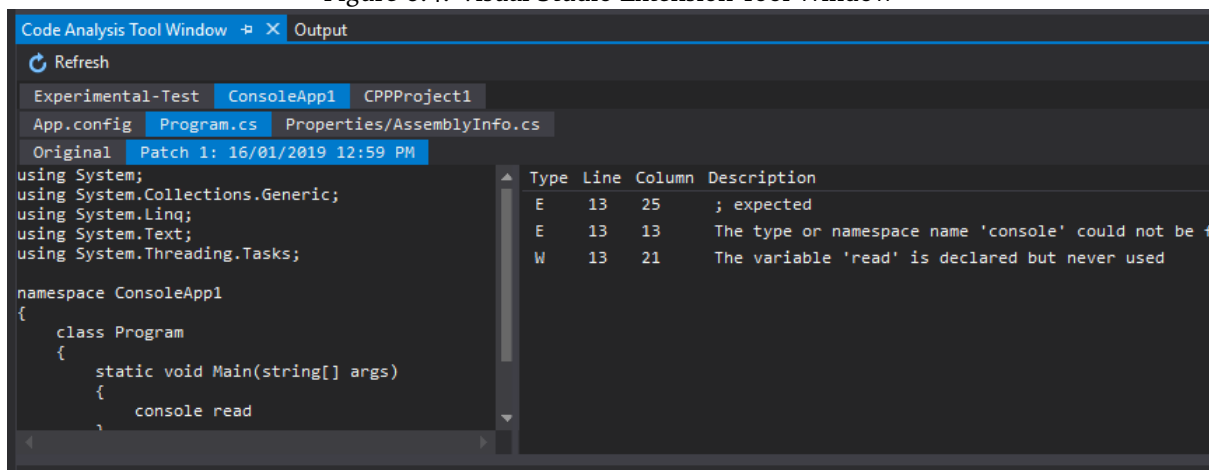
Figure 6.3: *Visual Studio* Output Window

Figure 6.4: Visual Studio Extension Tool Window



6.3 Development process and implementation details

This section describes the process used in the development of the artefact, listing and explaining the tools used for each software component. Further, it details how these components were implemented; explaining the choices of software frameworks and services used.

The programming part of the development process commenced, primarily, by studying existing documentation on *VSIX* (*Visual Studio* extensions) development (Microsoft, 2017c). As the extension consists of multiple components, each was worked on independently. Following this, all of the components were combined and linked together— for example integrating the cloud services into the extension; migrating from local development to production resources.

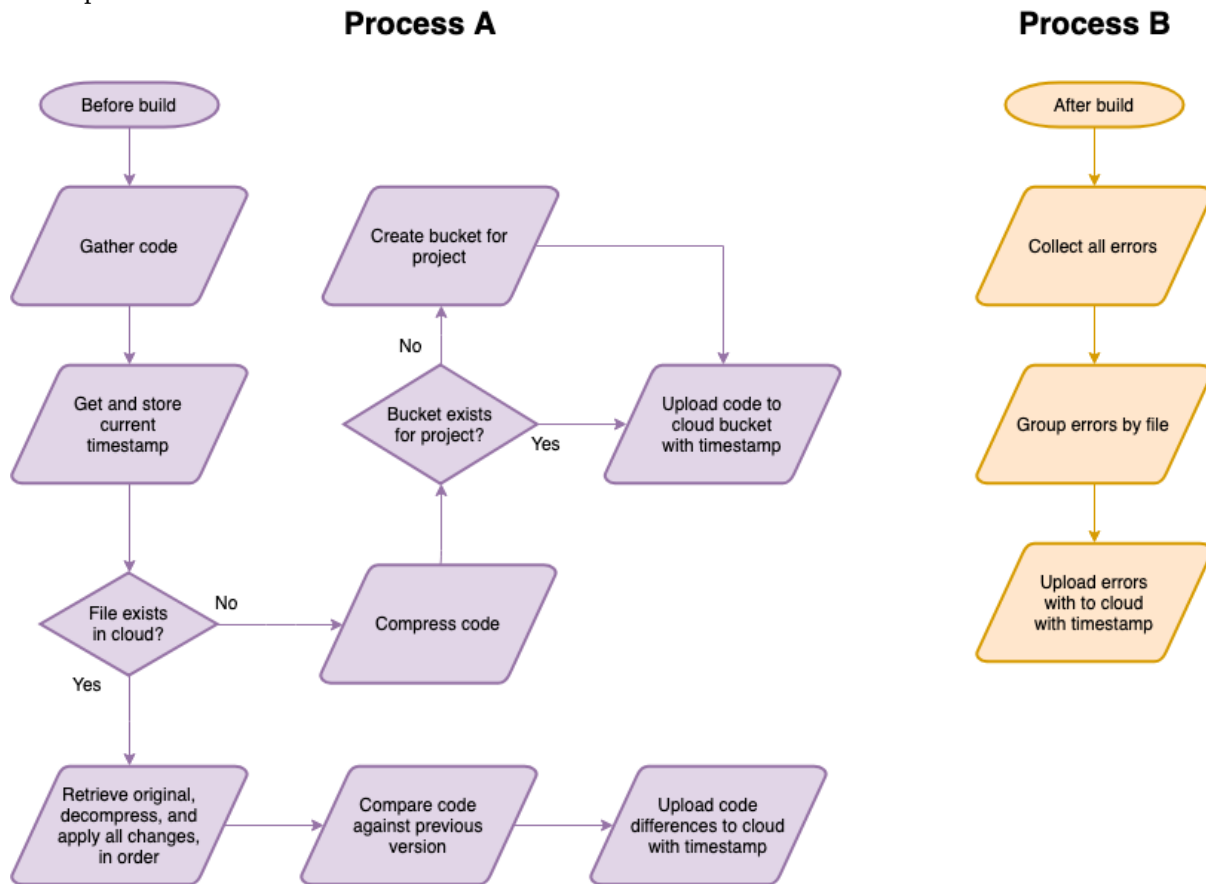
To provide simple and well-defined access to the data from outside of the *Visual Studio* extension, a *RESTful* API was created utilising the *Google Cloud Functions* platform (hereinafter referred to as *Cloud Functions*). The API itself was developed using the *Node.js* run-time environment for the *JavaScript* programming language (Node.js Foundation, undated). The primary reason for choosing to use this environment is—as well as the aforementioned reasons for using *Cloud Functions*—that *Node.js* is the primary language supported by *Cloud Functions* (Google Cloud, 2019c). An important distinction to make here is that the language in which the API runs is not relevant to how the data is able to be accessed (not limited to a particular language, such that it could not be used by a *Python* program, for example); instead, the API uses the standard *REST* software architecture (Filipe Ximenes and Flávio Juvenal, 2017), which is language independent. The data itself is represented in the *JSON* format, a standard, language-independent file format (Bray, 2017).

The functionality provided by the API primarily involves collecting a list of cloud storage buckets stored in *Google Cloud Storage* that belong to projects uploaded by the extension; either for every project, or limited to a particular project GUID. From each of the buckets found, all files are collected together—including both errors and differences—and combined into a *JSON* object array, with additional metadata added (for example timestamps or project GUIDs). The API provides read-only access; in implementation this means that the only HTTP (Fielding and Reschke, 2014) endpoint provided is that of *GET*, which provides data back to the user; it is impossible to add new data using the API— this is something that can be done only through the extension itself, which itself is authorised to upload.

A website was created to generate the figures and provide an at-a-glance overview of some sample statistics from the data uploaded by the extension. The website runs in the *Google App Engine* environment, a cloud hosted platform for web applications; in particular, supporting *Python*, with also the additional benefit of easily integrating with other *Google Cloud* services. This meant that it fit well into the existing architecture of the project and allowing for quick development and deployment; in this case, simply uploading (‘deploying’) the website with a simple command (Google Cloud, 2019a). The website—available from <https://lncn.ac/codeanalysis>—runs in the *Python* programming language, and uses the *Flask* library, a simple web framework. The figures themselves are generated by collecting data—gathered from the extension—from the API, before parsing the *JSON* data output by the API into various forms of statistics as required, before finally using the *Matplotlib* library to generate the images.

The primary component of the artefact—the *Visual Studio* extension follows a set process to collect

Figure 6.5: Flow chart showing how data is collected by the *Visual Studio* extension; process A takes place before a *Visual Studio* project is built; following that, process B happens immediately after a build is completed









its data, as shown in Figure 6.5. The general outline of this process is as follows:

1. Before a build takes place; gather the code contents from all files, in all projects within the solution.
2. Get and store the current timestamp.
3. If files were uploaded previously; retrieve the original file and all changes ('diffs') made to each file from the cloud and decompress them.
4. Apply all diffs, in the correct order, on top of the original version of each file as required– this is also known as 'patching' the files.
5. Compare the differences between the current version of each file against the 'patched' versions retrieved from the cloud.
6. If no previous versions of a file were found; compress the code file using the *BZip2* algorithm; if, further, there was no cloud bucket found for the current project GUID, create one.
7. Upload these differences to the cloud bucket with each file timestamped.
8. After the build finishes; collect all errors generated by the compiler and group them by file; attach the previously acquired timestamp to the errors.

For a sample of one *C#* code file, *Class2.cs*, the structure of the various files created by the extension within the project's cloud bucket are shown in Figure 6.6. As is evident from this structure, the original code file is uploaded as compressed (as this would be in most cases the largest file), with each *.diff* file timestamped following it (note that timestamps inherently follow an alphabetical order allowing for easy management), and where appropriate, an *.err* file with a timestamp corresponding to a *.diff* file (or if no corresponding diff, the original version).

Figure 6.6: Example of the contents of a cloud bucket for a single file in a project

<input type="checkbox"/> Name	Size	Type
<input type="checkbox"/>  Class2.cs.bz2	176 B	text/plain
<input type="checkbox"/>  Class2.cs.diff.2019-03-28T09:30:09	59 B	text/plain
<input type="checkbox"/>  Class2.cs.diff.2019-03-28T09:38:12	46 B	text/plain
<input type="checkbox"/>  Class2.cs.err.2019-01-23T13:42:46	143 B	text/plain
<input type="checkbox"/>  Class2.cs.err.2019-03-28T09:30:09	215 B	text/plain
<input type="checkbox"/>  Class2.cs.err.2019-03-28T09:38:12	2 B	text/plain

Once installed, the extension can generally be considered operational without any further input from the end-user (meaning no need to keep any login details, change any settings, etc.), aside from—in a case wherein the extension be released commercially—to install possible future software updates. Situations where a software update would be absolutely required include cases where the extension code would need to be changed to reflect back-end (server-side) changes; examples include: authentication details being changed; cloud services being changed; or, the scope of the data collected being changed. It is important to make clear, however, that these cases where change is needed are outside of the scope of this project (these changes are only theoretical possibilities, should the extension be turned into a commercial product), however, it is worth noting these cases for future reference.

As the extension installation can be considered ‘set-and-forget’, expected maintenance of the artefact is relatively low, although, an important aspect in considering the artefact for public release (i.e. outside of a test environment) would be to restrict the permissions of the ‘service account’ which is used to authenticate the *Visual Studio* extension. This service account is used to authenticate and allow access to the *Google Cloud* services from external software; in this case, the extension. This is a particularly important consideration, as it limits the scope of access to those cloud services to the minimum required for the operation of the extension (this is commonly known as the ‘principle of least privilege’, wherein, as Motiee et al. (2010) state, the most restrictive set of privileges is applied, while still allowing performing the required tasks); in case the security of the private key used for authentication becomes compromised. In this case, an example might be to limit the service account to allow only access to cloud buckets for reading and writing. It will also be important to maintain the API to ensure that it remains working for all requirements. Similarly, a limitation of the API in its current state is the limited speed of file retrieval from *Google Cloud Storage*. In this case, the API could be significantly improved through optimising the way in which it accesses data from storage, be it through some form of caching (storing copies of files that are commonly accessed) or otherwise.

6.4 Evaluation

The extension was tested by collecting data for a number of projects. Initial testing took place in the form of white-box testing through debugging the code itself in the *Visual Studio* instance used to develop the extension itself. This testing consisted of making sure that the various events were triggered as expected—by setting ‘break points’ in the code that trigger when a particular part of the code was reached. In other white-box testing, exception handling was tested; for example, testing that code executes correctly if there is no internet connection available; or, if the project is not saved to the local disk when a build is triggered—meaning in this case that it is not possible for the extension to retrieve information about files.

Other testing—forms of black-box testing—involved running *Visual Studio* with the extension installed, and going about the normal development process (change code, build, check errors, fix errors, repeat) for various *Visual Studio* projects. These projects contained code in both C# and C++; data from both languages were able to be collected as expected, including all the data required, namely for instance error codes and code text itself. These languages were chosen for this testing as they are supported by *Visual Studio* and, crucially, they require a build step; therefore, error outputs by *Visual Studio*—

and for additional reference and checking, these error codes are also available online (Microsoft, 2015). This form of testing is also used when running the extension in production— as it simply ensures that the extension performs its task—namely collecting data—correctly, without the deeper understanding of ‘how’, or consideration of exactly what is happening.

7. Project Conclusion

Overall, the artefact was considered to meet the requirements as set out by the client for the time-frame provided; this was confirmed through a meeting with the client upon ceasing the main phase of development for the artefact. In this meeting possible improvements for the artefact in future iterations—should the artefact be continued further beyond the project itself—were discussed. These potential changes included for example to allow the user the ability to see some of the data that has been collected about them so far in a user-friendly way, integrated into the development environment (as opposed to through the website).

The project has proved, in the artefact, to be successful in collecting errors generated by *Visual Studio*, automatically, and importantly, being able to store them in the cloud. In addition, where required, code files were able to be compressed automatically before being uploaded to cloud storage, and, contrastingly, decompressed as required—because of this, storage requirements were reduced. As has been previously described, data has been successfully gathered in cloud storage for a number of projects in both *C#* and *C++*; including cloud resources scaling, and the extension being able to create cloud resources as required (i.e. for each project, creating a new bucket). Similarly, the functionality of finding as well as storage of code differences over time were able to be stored in an efficient way, and, as demonstrated previously, able to be visualised; as in the ‘Tool Window’ (Figure 6.4) component of the extension.

Of particular note, the extension meets expectations in terms of functionality, meaning it can be considered able to operate autonomously (i.e. automatically, without user input) after being installed in *Visual Studio*. By being able to operate automatically, the extension meets requirements for collecting data from various projects, while also making it suitable to, as described previously, collect data in, for example, a learning environment.

As per the requirements, the structure of the data can be considered suitable to achieve its task, and in its raw state (without semantic meaning), it holds enough information to allow for its use in further analysis. Despite this, it can be considered difficult to evaluate the specific requirements of the data format, as the focus primarily revolved around what was needed to be stored, instead of specifying how it be stored; something that could be considered largely up to the individual implementation (instead, data can be transformed to the format as required, discussed earlier).

Further to this, as has been proved already, it is possible to use the data in various ways, as demonstrated in the statistical analysis through the website. Further, the data has implications for use in a wide variety of fields, for example, in teaching, understanding how students progress over time and which particular errors they face can be helpful in directing teaching efforts. A frequently mentioned example of how the data might be used is in machine learning, which has partially been discussed in a previous paper (Castro-Wunsch et al., 2017). As mentioned previously, it could be possible to use data with machine learning techniques to make suggestions to fix common errors as they occur; for example, a *Visual Studio* extension might be able to determine which parts of the code are causing a particular error (given more context) and suggest a potential fix, based on changes made by users in the past for similar circumstances.

8. Reflective Analysis

This section gives a reflection on the development process of the artefact; it provides a summary of the key things that went well and which didn't go as well as was planned, as well as what could have been done differently. Further, the section reflects on what could have been done in addition to what has already been done, given there was more time available to the project.

In general, the project development process went as planned—although the first few weeks were relatively slow in progress due to the learning process involved in creating a *Visual Studio* extension. The choice of tools worked well and meant that most of the development process was smooth; although one of the biggest hurdles encountered was the authentication setup with cloud resources, involving a sub-optimal integration of the authentication—i.e. embedding the key directly in the program rather than a more streamlined process; an alternative might have been to add this functionality to the API, given more time.

As the agile software development process was used throughout, it was possible to evaluate the progress of the development process semi-regularly throughout the process. Most commonly, evaluation took place in the form of feedback at the end of each agile 'sprint'; that is to say, after a number of tasks were completed, they were checked to meet the client's requirements, adjusting the project as necessary as it progressed.

A major part of the design process involved researching how *Visual Studio* is laid out and how to fit the extension's style into something similar to that used in that existing layout. Possibly, given more time, a better understanding of how *Visual Studio* allows for extending its layout could have been researched, as, at present, the extension relies on retrieving simple attributes such as the common colours used, rather than using a pre-defined 'theme', for example. As well as this, some improvements could have been made to the user experience aspects of the extension; to name an example, providing better user feedback— as currently the extension relies on all output being in the form of a simple text-only output. A better method might have been to display some of the ongoing information (e.g. upload in progress) using a status bar, but this was not implemented due to its additional complexity clashing with the time constraints.

Although the artefact is considered 'complete', a number of changes could be made to improve the quality of its implementation code. This is due to the fact that, as documentation is limited for the process of developing a *Visual Studio*, some shortcuts ('hacks') were used—for example using global variables, generally considered bad practise in large-scale programs—to get the extension to 'work' in a relatively short amount of time— these shortcuts ideally should be replaced by finding the canonical methods for implementing such pieces of code. Similarly, due to time constraints, testing and error handling are more limited than would be expected for a 'commercial' product, meaning there might be bugs that affect the stability or functionality of the artefact in certain scenarios which could not be considered previously due to this limited time. Despite the aforementioned points and potential amendments that could be made, the functionality of the artefact met expectations and requirements of the client.

Data collected by the extension so far has been useful in understanding some of the common errors generated by *Visual Studio* in the form of some basic statistical analysis; although with more data the understanding of the data as a whole can improve, especially had more time been allowed for the extension to operate— e.g. if its development had been completed earlier with more time for testing in operation. As well as this, it would have been even better to see how the data could be used in machine learning by trialling some machine learning methods. By doing so, it could also have been possible to understand what features of the data were considered relevant and which were less-so as important; this could have helped to restructure the data as required relatively early on in the development process. Further, it could have provided better understanding of how the data is interpreted by a machine learning algorithm, i.e. to better understand what components of the data have which relations to each other, something that had to be considered manually for the purposes of creating the simple statistical analysis.

mentioned prior. With this better understanding, it could have been possible to better understand how such data could be used and visualised in a more meaningful way than simply generating statistics in the form of plots.

Throughout the process of developing the artefact, some useful insight was gained for the process of understanding and interpreting a codebase for a project with little documentation (in this case, *Visual Studio* extensions). Further to this, additional understanding of how errors are gathered by *Visual Studio* and, in general, how and what they are formed of, as well as how they are generated, helped in understanding the development process even further.

9. References

- Ahadi, A. et al. (2015). Exploring Machine Learning Methods to Automatically Identify Students in Need of Assistance. *Proceedings of the Eleventh Annual International Conference on International Computing Education Research - ICER '15*. The Eleventh Annual International Conference. Omaha, Nebraska, USA: 121–130.
- Amazon (undated). *Cloud Object Storage — Store & Retrieve Data Anywhere — Amazon Simple Storage Service*. Available from <https://aws.amazon.com/s3/> [accessed 3rd April 2019].
- Becker, B. A. (2016a). A New Metric to Quantify Repeated Compiler Errors for Novice Programmers. *Proceedings of the 2016 ACM Conference on Innovation and Technology in Computer Science Education - ITiCSE '16*. The 2016 ACM Conference. Arequipa, Peru: 296–301.
- Becker, B. A. (2016b). An Effective Approach to Enhancing Compiler Error Messages. *Proceedings of the 47th ACM Technical Symposium on Computing Science Education - SIGCSE '16*. The 47th ACM Technical Symposium. Memphis, Tennessee, USA: 126–131.
- Bell, T. (1997). *The Canterbury Corpus*. Available from <http://corpus.canterbury.ac.nz/descriptions/#cantbrbry> [accessed 29th March 2019].
- Bhat, W. A. (2018). Is a Data-Capacity Gap Inevitable in Big Data Storage? *Computer* 51(9), 54–62.
- Bray, T. (2017). *The JavaScript Object Notation (JSON) Data Interchange Format*. Available from <https://tools.ietf.org/html/rfc8259> [accessed 28th March 2019].
- Buckley, J. et al. (2005). Towards a Taxonomy of Software Change. *Journal of Software Maintenance and Evolution: Research and Practice* 17(5), 309–332.
- Campbell, J. C., Hindle, A. and Amaral, J. N. (2014). Syntax Errors Just Aren't Natural: Improving Error Reporting with Language Models. *Proceedings of the 11th Working Conference on Mining Software Repositories - MSR 2014*. The 11th Working Conference. Hyderabad, India: 252–261.
- Castro-Wunsch, K., Ahadi, A. and Petersen, A. (2017). Evaluating Neural Networks as a Method for Identifying Students in Need of Assistance. *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education - SIGCSE '17*. The 2017 ACM SIGCSE Technical Symposium. Seattle, Washington, USA: 111–116.
- Fielding, R. and Reschke, J. (2014). *Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content*. Available from <https://tools.ietf.org/html/rfc7231> [accessed 28th March 2019].
- Filipe Ximenes and Flávio Juvenal (2017). *How to Design a RESTful API Architecture from a Human-Language Spec - O'Reilly Media*. Available from <https://www.oreilly.com/learning/how-to-design-a-restful-api-architecture-from-a-human-language-spec> [accessed 28th March 2019].
- Fowler, M. and Scott, K. (2000). *UML Distilled: A Brief Guide to the Standard Object Modeling Language*. 2nd ed. Object Technology Series. Reading, Mass: 185 pp.
- GitLab (undated). *GitLab Pricing*. Available from <https://about.gitlab.com/pricing/> [accessed 4th April 2019].
- Google Cloud (2019a). *Deploying the Application — App Engine Standard Environment for Python*. Available from <https://cloud.google.com/appengine/docs/standard/python/getting-started/deploying-the-application> [accessed 28th March 2019].
- Google Cloud (2019b). *HTTP Functions — Cloud Functions Documentation*. Available from <https://cloud.google.com/functions/docs/writing/http> [accessed 28th March 2019].
- Google Cloud (2019c). *Writing Cloud Functions — Cloud Functions Documentation*. Available from <https://cloud.google.com/functions/docs/writing/> [accessed 28th March 2019].
- Hamilton, J. (2003). Language Integration in the Common Language Runtime. *ACM SIGPLAN Notices* 38(2), 19.
- Hashem, I. A. T. et al. (2015). The Rise of “Big Data” on Cloud Computing: Review and Open Research Issues. *Information Systems* 47, 98–115.

- Jadud, M. C. (2005). A First Look at Novice Compilation Behaviour Using BlueJ. *Computer Science Education* 15(1), 25–40.
- Johnson, H. A. (2017). Trello. *Journal of the Medical Library Association : JMLA* 105(2), 209–211. pmid: null.
- Khan, A. I., Qureshi, M. R. J. and Khan, U. A. (2012). A Comprehensive Study of Commonly Practiced Heavy & Light Weight Software Methodologies. arXiv: 1202.2514 [cs].
- Kreuger, M. (undated). *SharpZipLib*. Available from <http://icsharpcode.github.io/SharpZipLib/> [accessed 30th March 2019].
- Microsoft (2015). *C# Compiler Errors*. Available from <https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/compiler-messages/> [accessed 1st April 2019].
- Microsoft (2017a). *Create Your First Extension: Hello World*. Available from <https://docs.microsoft.com/en-us/visualstudio/extensibility/extensibility-hello-world?view=vs-2017> [accessed 6th November 2018].
- Microsoft (2017b). *ErrorItem Interface (EnvDTE80)*. Available from <https://docs.microsoft.com/en-us/dotnet/api/envdte80.erroritem> [accessed 6th April 2019].
- Microsoft (2017c). *Starting to Develop Visual Studio Extensions - Visual Studio*. Available from <https://docs.microsoft.com/en-us/visualstudio/extensibility/starting-to-develop-visual-studio-extensions> [accessed 6th November 2018].
- Microsoft (undated[a]). *Blob Storage — Microsoft Azure*. Available from <https://azure.microsoft.com/en-gb/services/storage/blobs/> [accessed 3rd April 2019].
- Microsoft (undated[b]). *Guid Struct (System) — Microsoft Docs*. Available from <https://docs.microsoft.com/en-us/dotnet/api/system.guid?view=netframework-4.7.2> [accessed 29th March 2019].
- Miller, G. A. (1956). The Magical Number Seven, plus or Minus Two: Some Limits on Our Capacity for Processing Information. *Psychological review* 63(2), 81.
- Motiee, S., Hawkey, K. and Beznosov, K. (2010). Investigating User Account Control Practices. *Proceedings of the 28th of the International Conference Extended Abstracts on Human Factors in Computing Systems - CHI EA '10*. The 28th of the International Conference Extended Abstracts. Atlanta, Georgia, USA: 4129.
- Muñoz-Escóí, F. and Bernabéu-Aubán, J. (2017). A Survey on Elasticity Management in PaaS Systems. *Computing* 99(7), 617–656.
- Node.js Foundation (undated). *Node.js*. Available from <https://nodejs.org/> [accessed 28th March 2019].
- Nyström, A. (2011). Agile Solo-Defining and Evaluating an Agile Software Development Process for a Single Software Developer (Journal Article).
- Pawar, V. J., Kharat, K. D. and Pardeshi, S. (2016). Structuring Cloud Computing Using Big Data Analytics Solution: A Survey. *2016 International Conference on Communication and Electronics Systems (ICCES)*. 2016 International Conference on Communication and Electronics Systems (ICCES), 1–6.
- Polk, R. (2011). Agile and Kanban in Coordination. *2011 Agile Conference*. 2011 Agile Conference, 263–268.
- Raza, A. et al. (2017). Impact of Story Point Estimation on Product Using Metrics in Scrum Development Process. *International Journal of Advanced Computer Science and Applications* 8(4).
- Shanmugasundaram, S. and Lourdasamy, R. (2011). A Comparative Study Of Text Compression Algorithms. 1, 9.
- Spinellis, D. (2012). Git. *IEEE Software* 29(3), 100–101.
- Sutherland, J. et al. (2007). Distributed Scrum: Agile Project Management with Outsourced Development Teams. *2007 40th Annual Hawaii International Conference on System Sciences (HICSS'07)*. 2007 40th Annual Hawaii International Conference on System Sciences (HICSS'07), 274a–274a.
- Tak, B. C., Urgaonkar, B. and Sivasubramaniam, A. (2013). Cloudy with a Chance of Cost Savings. *IEEE Transactions on Parallel and Distributed Systems* 24(6), 1223–1233.
- Thorpe, L. (2018). Project Proposal.
- Trello (2019). *Trello*. Available from <https://trello.com> [accessed 13th January 2019].
- Troelsen, A. W. (2005). *Pro C# 2005 and the .NET 2.0 Platform*. 3rd ed. OCLC: ocm61770542. Berkeley, Calif:

A. Appendices

Appendix A.1: The code file `fields.c` from Canterbury Corpus, which is used to test the performance of lossless compression algorithms

```
#ifndef lint
static char Rcs_Id[] =
    "$Id: fields.c,v 1.7 1994/01/06 05:26:37 geoff Exp $";
#endif

/*
 * $Log: fields.c,v $
 * Revision 1.7 1994/01/06 05:26:37 geoff
 * Get rid of all references to System V string routines, for portability
 * (sigh).
 *
 * Revision 1.6 1994/01/05 20:13:43 geoff
 * Add the maxf parameter
 *
 * Revision 1.5 1994/01/04 02:40:21 geoff
 * Make the increments settable (field_line_inc and field_field_inc).
 * Add support for the FLD_NOSHRINK flag.
 *
 * Revision 1.4 1993/09/27 17:48:02 geoff
 * Fix some lint complaints and some parenthesization errors.
 *
 * Revision 1.3 1993/09/09 01:11:11 geoff
 * Add a return value to fieldwrite. Add support for backquotes and for
 * unstripped backslashes.
 *
 * Revision 1.2 1993/08/26 00:02:50 geoff
 * Fix a stupid null-pointer bug
 *
 * Revision 1.1 1993/08/25 21:32:05 geoff
 * Initial revision
 *
 */

#include <stdio.h>
#include "config.h"
#include "fields.h"

field_t * fieldread P ((FILE * file, char * delims,
    int flags, int maxf));
    /* Read a line with fields from a file */
field_t * fieldmake P ((char * line, int allocated, char * delims,
    int flags, int maxf));
    /* Make a field structure from a line */
static field_t * fieldparse P ((field_t * fieldp, char * line, char * delims,
```

```
        int flags, int maxf));
    /* Parse the fields in a line */
static int  fieldbackch P ((char * str, char ** out, int strip));
    /* Process backslash sequences */
int  fieldwrite P ((FILE * file, field_t * fieldp, int delim));
    /* Write a line with fields to a file */
void  fieldfree P ((field_t * fieldp));
    /* Free a field returned by fieldread */

unsigned int  field_field_inc = 20; /* Increment to increase # fields by */
unsigned int  field_line_inc = 512; /* Incr to increase line length by */

#ifdef USG
#define strchr  index
#endif /* USG */

extern void free ();
extern char * malloc ();
extern char * realloc ();
extern char * strchr ();
extern int  strlen ();

/*
 * Read one line of the given file into a buffer, break it up into
 * fields, and return them to the caller.  The field_t structure
 * returned must eventually be freed with fieldfree.
 */
field_t * fieldread (file, delims, flags, maxf)
    FILE *    file; /* File to read lines from */
    char *    delims; /* Characters to use for field delimiters */
    int      flags; /* Option flags; see fields.h */
    int      maxf; /* Maximum number of fields to parse */
{
    register char * linebuf; /* Buffer to hold the line read in */
    int      linemax; /* Maximum line buffer size */
    int      linesize; /* Current line buffer size */

    linebuf = (char *) malloc (field_line_inc);
    if (linebuf == NULL)
return NULL;
    linemax = field_line_inc;
    linesize = 0;
    /*
     * Read in the line.
     */
    while (fgets (&linebuf[linesize], linemax - linesize, file)
        != NULL)
    {
        linesize += strlen (&linebuf[linesize]);
        if (linebuf[linesize - 1] == '\n')
            break;
        else
        {
            linemax += field_line_inc;
            linebuf = (char *) realloc (linebuf, linemax);
            if (linebuf == NULL)
return NULL;
        }
    }
}
```

```
    }
    if (linesize == 0)
    {
        free (linebuf);
        return NULL;
    }
    return fieldmake (linebuf, 1, delims, flags, maxf);
}

field_t * fieldmake (line, allocated, delims, flags, maxf)
char *    line; /* Line to make into a field structure */
int       allocated; /* NZ if line allocated with malloc */
char *    delims; /* Characters to use for field delimiters */
int       flags; /* Option flags; see fields.h */
int       maxf; /* Maximum number of fields to parse */
{
    register field_t * fieldp; /* Structure describing the fields */
    int     linesize; /* Current line buffer size */

    fieldp = (field_t *) malloc (sizeof (field_t));
    if (fieldp == NULL)
        return NULL;
    fieldp->nfields = 0;
    fieldp->linebuf = allocated ? line : NULL;
    fieldp->fields = NULL;
    fieldp->hadnl = 0;
    linesize = strlen (line);
    if (line[linesize - 1] == '\n')
    {
        line[--linesize] = '\0';
        fieldp->hadnl = 1;
    }

    /*
     * Shrink the line buffer if necessary.
     */
    if (allocated && (flags & FLD_NOSHRINK) == 0)
    {
        line = fieldp->linebuf =
            (char *) realloc (fieldp->linebuf, linesize + 1);
        if (fieldp->linebuf == NULL)
        {
            fieldfree (fieldp);
            return NULL;
        }
    }

    return fieldparse (fieldp, line, delims, flags, maxf);
}

static field_t * fieldparse (fieldp, line, delims, flags, maxf)
register field_t * fieldp; /* Field structure to parse into */
register char * line; /* Line to be parsed */
char *    delims; /* Characters to use for field delimiters */
int       flags; /* Option flags; see fields.h */
int       maxf; /* Maximum number of fields to parse */
{
    int     fieldmax; /* Max size of fields array */
    char *  lineout; /* Where to store xlated char in line */
    char    quote; /* Quote character in use */

```

```
    fieldp->nfields = 0;
    fieldmax =
        (maxf != 0 && maxf < field_field_inc) ? maxf + 2 : field_field_inc;
    fieldp->fields = (char **) malloc (fieldmax * sizeof (char *));
    if (fieldp->fields == NULL)
    {
        fieldfree (fieldp);
        return NULL;
    }
    if ((flags
    & (FLD_SHQUOTES | FLD_SNGQUOTES | FLD_BACKQUOTES | FLD_DBLQUOTES))
        == FLD_SHQUOTES)
    flags |= FLD_SNGQUOTES | FLD_BACKQUOTES | FLD_DBLQUOTES;
    while (1)
    {
        if (flags & FLD_RUNS)
        {
            while (*line != '\0' && strchr (delims, *line) != NULL)
                line++;
            /* Skip runs of delimiters */
            if (*line == '\0')
                break;
        }
        fieldp->fields[fieldp->nfields] = lineout = line;
        /*
        * Skip to the next delimiter. At the end of skipping, "line" will
        * point to either a delimiter or a null byte.
        */
        if (flags
            & (FLD_SHQUOTES | FLD_SNGQUOTES | FLD_BACKQUOTES
              | FLD_DBLQUOTES | FLD_BACKSLASH))
        {
            while (*line != '\0')
            {
                if (strchr (delims, *line) != NULL)
                    break;
                else if (((flags & FLD_SNGQUOTES) && *line == '\'' )
                    || ((flags & FLD_BACKQUOTES) && *line == '\"')
                    || ((flags & FLD_DBLQUOTES) && *line == '\"'))
                {
                    if ((flags & FLD_SHQUOTES) == 0
                        && line != fieldp->fields[fieldp->nfields])
                        quote = '\0';
                    else
                        quote = *line;
                }
            }
            else
                quote = '\0';
            if (quote == '\0')
            {
                if (*line == '\\' && (flags & FLD_BACKSLASH))
                {
                    line++;
                    if (*line == '\0')
                        break;
                }
                line += fieldbackch (line, &lineout,
                    flags & FLD_STRIPQUOTES);
            }
        }
```

```
        else
            *lineout++ = *line++;
    }
    else
    {
        /* Process quoted string */
        if ((flags & FLD_STRIPQUOTES) == 0)
            *lineout++ = quote;
        ++line;
        while (*line != '\0')
        {
            if (*line == quote)
            {
                if ((flags & FLD_STRIPQUOTES) == 0)
                    *lineout++ = quote;
                line++; /* Go on past quote */
                if ((flags & FLD_SHQUOTES) == 0)
                {
                    while (*line != '\0'
                        && strchr (delims, *line) == NULL)
                        line++; /* Skip to delimiter */
                }
                break;
            }
            else if (*line == '\\')
            {
                if (flags & FLD_BACKSLASH)
                {
                    line++;
                    if (*line == '\0')
                        break;
                }
                else
                {
                    line += fieldbackch (line, &lineout,
                        flags & FLD_STRIPQUOTES);
                }
            }
            else
            {
                *lineout++ = '\\';
                if (++line == '\0')
                    break;
                *lineout++ = *line;
            }
        }
        else
            *lineout++ = *line++;
    }
}
}
}
else
{
    while (*line != '\0' && strchr (delims, *line) == NULL)
        line++; /* Skip to delimiter */
    lineout = line;
}
fieldp->nfields++;
if (*line++ == '\0')
    break;
```

```
    if (maxf != 0 && fieldp->nfields > maxf)
        break;
    *lineout = '\0';
    if (fieldp->nfields >= fieldmax)
    {
        fieldmax += field_field_inc;
        fieldp->fields =
            (char **) realloc (fieldp->fields, fieldmax * sizeof (char *));
        if (fieldp->fields == NULL)
        {
            fieldfree (fieldp);
            return NULL;
        }
    }
    /*
     * Shrink the field pointers and return the field structure.
     */
    if ((flags & FLD_NOSHRINK) == 0 && fieldp->nfields >= fieldmax)
    {
        fieldp->fields = (char **) realloc (fieldp->fields,
            (fieldp->nfields + 1) * sizeof (char *));
        if (fieldp->fields == NULL)
        {
            fieldfree (fieldp);
            return NULL;
        }
    }
    fieldp->fields[fieldp->nfields] = NULL;
    return fieldp;
}

static int fieldbackch (str, out, strip)
    register char * str;    /* First char of backslash sequence */
    register char ** out;   /* Where to store result */
    int strip;             /* NZ to convert the sequence */
{
    register int ch;        /* Character being developed */
    char * origstr;         /* Original value of str */

    if (!strip)
    {
        *(*out)++ = '\\';
        if (*str != 'x' && *str != 'X' && (*str < '0' || *str > '7'))
        {
            *(*out)++ = *str;
            return *str != '\0';
        }
    }
    switch (*str)
    {
    case '\0':
        *(*out)++ = '\0';
        return 0;
    case 'a':
        *(*out)++ = '\007';
        return 1;
    case 'b':
```



```
       >(*out)++ = '\b';
        return 1;
    case 'f':
        (*out)++ = '\f';
        return 1;
    case 'n':
        (*out)++ = '\n';
        return 1;
    case 'r':
        (*out)++ = '\r';
        return 1;
    case 'v':
        (*out)++ = '\v';
        return 1;
    case 'X':
    case 'x':
        /* Hexadecimal sequence */
        origstr = str++;
        ch = 0;
        if (*str >= '0' && *str <= '9')
            ch = *str++ - '0';
        else if (*str >= 'a' && *str <= 'f')
            ch = *str++ - 'a' + 0xa;
        else if (*str >= 'A' && *str <= 'F')
            ch = *str++ - 'A' + 0xa;
        if (*str >= '0' && *str <= '9')
            ch = (ch << 4) | (*str++ - '0');
        else if (*str >= 'a' && *str <= 'f')
            ch = (ch << 4) | (*str++ - 'a' + 0xa);
        else if (*str >= 'A' && *str <= 'F')
            ch = (ch << 4) | (*str++ - 'A' + 0xa);
        break;
    case '0':
    case '1':
    case '2':
    case '3':
    case '4':
    case '5':
    case '6':
    case '7':
        /* Octal sequence */
        origstr = str;
        ch = *str++ - '0';
        if (*str >= '0' && *str <= '7')
            ch = (ch << 3) | (*str++ - '0');
        if (*str >= '0' && *str <= '7')
            ch = (ch << 3) | (*str++ - '0');
        break;
    default:
        (*out)++ = *str;
        return 1;
}
    if (strip)
    {
        (*out)++ = ch;
        return str - origstr;
    }
    else
```

```
{
    for (ch = 0; origstr < str; ch++)
        *(*out)++ = *origstr++;
    return ch;
}

int fieldwrite (file, fieldp, delim)
    FILE *    file; /* File to write to */
    register field_t *  fieldp; /* Field structure to write */
    int      delim; /* Delimiter to place between fields */
{
    int      error; /* NZ if an error occurs */
    register int  fieldno; /* Number of field being written */

    error = 0;
    for (fieldno = 0; fieldno < fieldp->nfields; fieldno++)
    {
        if (fieldno != 0)
            error |= putc (delim, file) == EOF;
        error |= fputs (fieldp->fields[fieldno], file) == EOF;
    }
    if (fieldp->hadnl)
        error |= putc ('\n', file) == EOF;
    return error;
}

void fieldfree (fieldp)
    register field_t *  fieldp; /* Field structure to free */
{
    if (fieldp == NULL)
        return;
    if (fieldp->linebuf != NULL)
        free ((char *) fieldp->linebuf);
    if (fieldp->fields != NULL)
        free ((char *) fieldp->fields);
    free ((char *) fieldp);
}
```