Callum Yates, 22042944

Y042944M

Abstract

A report detailing the research and development of a language-agnostic AI-assisted code review tool that analyses source code for adherence to design patterns and coding standards

DPAAT: Design Pattern Adhearance Analysis TOOL

Final Year Project

# Table of Contents

[Table of Contents 1](#_Toc197096653)

[Table of Figures 4](#_Toc197096654)

[1 Introduction 4](#_Toc197096655)

[1.1 Background 4](#_Toc197096656)

[1.2 Reason for Research 5](#_Toc197096657)

[1.3 Aims and objectives 5](#_Toc197096658)

[1.4 Outcomes and Deliverables 6](#_Toc197096659)

[1.5 Ethics 6](#_Toc197096660)

[1.6 Risks 6](#_Toc197096661)

[1.7 How the Report is Organised 6](#_Toc197096662)

[2 Methodologies 7](#_Toc197096663)

[2.1 Introduction 7](#_Toc197096664)

[2.1.1 What is a methodology? 7](#_Toc197096665)

[2.1.2 What are methods? 7](#_Toc197096666)

[2.1.3 Why use a methodology? 7](#_Toc197096667)

[2.2 Software methodologies 8](#_Toc197096668)

[2.2.1 Agile 8](#_Toc197096669)

[2.2.2 Waterfall 9](#_Toc197096670)

[2.2.3 Test Driven Development 9](#_Toc197096671)

[2.2.4 Chosen Methodology 9](#_Toc197096672)

[2.3 Research Onion 10](#_Toc197096673)

[2.3.1 Philosophy 11](#_Toc197096674)

[2.3.2 Approaches 12](#_Toc197096675)

[2.3.3 Strategies 13](#_Toc197096676)

[2.3.4 Choices 14](#_Toc197096677)

[2.3.5 Time Horizons 14](#_Toc197096678)

[2.3.6 Techniques 15](#_Toc197096679)

[2.4 Project Management Methods 15](#_Toc197096680)

[2.5 Project Management 16](#_Toc197096681)

[2.5.1 Shortcut 16](#_Toc197096682)

[2.5.2 Logbook 16](#_Toc197096683)

[2.5.3 Version control 16](#_Toc197096684)

[3 Domain Research 16](#_Toc197096685)

[3.1 Design patterns 16](#_Toc197096686)

[3.1.1 Standardisation Across Teams 16](#_Toc197096687)

[3.1.2 Improved Code Readability and Collaboration 17](#_Toc197096688)

[3.1.3 Enhanced Scalability and Extensibility 17](#_Toc197096689)

[3.2 Coding best practices 17](#_Toc197096690)

[3.2.1 Adherence to Consistent Naming Conventions 18](#_Toc197096691)

[3.2.2 Avoiding Code Duplication (DRY principle) 18](#_Toc197096692)

[3.2.3 Ensuring Code is Self-documenting. 18](#_Toc197096693)

[3.2.4 SOLID Principles 18](#_Toc197096694)

[3.3 Error Handling 20](#_Toc197096695)

[3.3.1 Use Specific Exceptions 20](#_Toc197096696)

[3.3.2 Centralised Error Logging 20](#_Toc197096697)

[3.3.3 Graceful Degradation 20](#_Toc197096698)

[3.3.4 Fail-Fast vs. Fail-Safe 21](#_Toc197096699)

[3.3.5 Avoiding Fail-Silent Systems: 22](#_Toc197096700)

[3.3.6 Tools for code quality assurance 22](#_Toc197096701)

[3.3.7 Challenges in Enforcing Standards 24](#_Toc197096702)

[4 Evaluation of domain research 26](#_Toc197096703)

[4.1 Summary of findings 26](#_Toc197096704)

[4.2 Aim of the solution 26](#_Toc197096705)

[4.3 Requirements Inferred from Domain Research 26](#_Toc197096706)

[4.3.1 Language-Agnostic Analysis 27](#_Toc197096707)

[4.3.2 Design Pattern Conformity Analysis 27](#_Toc197096708)

[4.3.3 Flexible Rule Application 27](#_Toc197096709)

[4.3.4 Real-Time or Near Real-Time Performance 27](#_Toc197096710)

[4.3.5 Context-Aware Feedback 28](#_Toc197096711)

[4.3.6 Integration with Existing Development Tools 28](#_Toc197096712)

[4.3.7 Usability and Accessibility 28](#_Toc197096713)

[5 Technology Research 28](#_Toc197096714)

[5.1 Initial Considerations of the Proposed Solution 28](#_Toc197096715)

[5.2 Evaluating Large Language Models for Code Analysis 29](#_Toc197096716)

[5.2.1 Overview Comparison of Current Models 29](#_Toc197096717)

[5.2.2 Hallucination risks 31](#_Toc197096718)

[5.3 Backend Development Framework 32](#_Toc197096719)

[5.3.1 Python 32](#_Toc197096720)

[5.3.2 File Handling 33](#_Toc197096721)

[5.3.3 Unit Testing and Validation Strategy 33](#_Toc197096722)

[5.3.4 Node.js 34](#_Toc197096723)

[5.3.5 File Handling 35](#_Toc197096724)

[5.3.6 Unit Testing and Validation Strategy 35](#_Toc197096725)

[5.3.7 Evaluation Against Project Requirements 36](#_Toc197096726)

[5.4 Automated Analysis and Workflow Integration 37](#_Toc197096727)

[5.4.1 Manual repository analysis 37](#_Toc197096728)

[5.4.2 Webhook triggered feedback 37](#_Toc197096729)

[5.5 Risk Acknowledgements and Limitations 38](#_Toc197096730)

[6 Design 39](#_Toc197096731)

[6.1 Requirements Summary and Design Input 39](#_Toc197096732)

[6.2 System Design 40](#_Toc197096733)

[6.3 Implementation 40](#_Toc197096734)

[6.4 Testing 40](#_Toc197096735)

[6.5 Deployment and Maintenance 41](#_Toc197096736)

[7 Project testing 41](#_Toc197096737)

[7.1 Testing Methodology 41](#_Toc197096738)

[7.2 Unit Testing 41](#_Toc197096739)

[7.3 Validation Testing 42](#_Toc197096740)

[7.4 Test Results and Evaluation 42](#_Toc197096741)

[7.5 Limitations of Testing 42](#_Toc197096742)

[8 Critical Reflection 43](#_Toc197096743)

[8.1 Project Success 43](#_Toc197096744)

[8.1.1 Artefact Functionality 43](#_Toc197096745)

[8.1.2 Usability and Developer Experience 43](#_Toc197096746)

[8.1.3 Extensibility 44](#_Toc197096747)

[8.2 Extent of Learning 44](#_Toc197096748)

[8.2.1 Technical Skills 44](#_Toc197096749)

[8.2.2 Research and Evaluation 44](#_Toc197096750)

[8.2.3 Project Management 45](#_Toc197096751)

[8.3 Future enhancements 45](#_Toc197096752)

[8.3.1 GitHub Integration 45](#_Toc197096753)

[8.3.2 Context aware feedback 45](#_Toc197096754)

[8.3.3 Performance Optimisation 46](#_Toc197096755)

[8.3.4 Career Influence 46](#_Toc197096756)

[8.3.5 Confidence and Growth 46](#_Toc197096757)

[8.3.6 Awareness of Industry Practices 47](#_Toc197096758)

[9 Conclusion 47](#_Toc197096759)

[10 Bibliography 48](#_Toc197096760)

# Table of Figures

[Figure 1: Research Onion (Saunders et al., 2019) 9](#_Toc196771847)

[Figure 2: Comparison of five research philosophical positions in business and management research (Saunders et al., 2019) 10](#_Toc196771848)

[Figure 3: Deduction, induction and abduction: from reason to research (Saunders et al., 2019) 11](#_Toc196771849)

# Introduction

## Background

The widespread integration of generative artificial intelligence (AI) tools, such as GitHub Copilot and ChatGPT, has significantly transformed contemporary software development practices. These systems offer substantial productivity gains, enabling rapid prototyping and accelerated implementation of routine programming tasks. However, recent studies have shown that code produced by such tools often lacks alignment with established software design principles, leading to diminished readability, maintainability, and long-term quality (Liu et al., 2024). This issue is not only confined to AI-generated code; developers at all levels frequently produce code that deviates from best practices due to time constraints, lack of experience, or inconsistent enforcement of standards. Consequently, there is an emerging need for tools that operate alongside these AI systems to review, interpret, and improve the quality of the code they generate, as well as that written by humans. This project addresses that gap by proposing an assistant capable of evaluating code quality and offering structured, context-aware feedback, with the aim of promoting adherence to design patterns and software engineering principles across development contexts.

## Reason for Research

The rationale for this project lies in the observed limitations of existing AI-assisted coding tools and their impact on software reliability. As AI technologies become an integral part of the development process, there is a pressing need for supplementary tools to mitigate the risks associated with their widespread use. This project not only addresses a relevant industry problem but also aligns with the objectives of a Computer Science degree by incorporating key concepts such as software engineering, code quality assurance, and tool development.

## Aims and objectives

The objectives of this project can be split into primary and secondary, primary being core functionalities that are essential, with secondaries being desirable but ultimately optional.

Primary Objectives:

1. Design and implement a tool that analyses source code for adherence to coding standards and best practices, focusing on readability, naming conventions, and structural clarity.
2. Provide actionable, developer-oriented feedback based on the analysis, to support improvement during active development.
3. Evaluate the effectiveness of the tool using selected code snippets that simulate real-world scenarios.

Secondary Objectives:

1. Investigate and trial user interface features to enhance accessibility and user experience.
2. Explore methods for supporting multiple programming languages or flexible rulesets.
3. Assess the feasibility of integrating version control systems (e.g. GitHub) for direct repository analysis.

## Outcomes and Deliverables

The project will deliver a software application that supports developers in assessing and improving the quality of their code. Key deliverables include:

* A functional application interface that allows developers to select code files or snippets for analysis.
* An analysis engine that reviews submitted code and outputs structured feedback aligned with recognised coding standards and development best practices.
* A mechanism for presenting this feedback to the user in a way that is meaningful, constructive, and actionable.
* Supporting documentation outlining the research and development process, rationale for design decisions, and testing methodology.

## Ethics

This project will be conducted in strict accordance with the University’s ethical guidelines. Where user feedback is collected, participants will be fully informed of their rights, including the ability to withdraw their data at any point during the study. Additionally, all collected data will be anonymised to protect user privacy and ensure confidentiality.

For a proportional review of the ethical considerations, refer to Appendix A.

## Risks

Refer to Appendix B for a detailed evaluation of the risks identified during the project planning phase.

## How the Report is Organised

This report is structured to guide the reader through the full lifecycle of the project, from initial motivation to final evaluation. Following this introduction, the report starts with discussing the methodologies that have informed the project, before conducting domain wide research on identifying the issues and solutions that exist surrounding code quality assurance. Next evaluating this research and using it to inform technology research, assessing tools and frameworks against the identified requirements.

The rest of the report focuses on the design, implementation, testing and reflection of the projects success, challenges, and learning outcomes.

# Methodologies

## Introduction

When performing any research, it is important to follow a research methodology to ensure the quality of the research (Saunders et al., 2019). This section details the selected research approach and data collection techniques.

### What is a methodology?

A methodology refers to the overarching framework or system of principles that guides the organisation, execution, and evaluation of a research or development project (Saunders et al., 2019). ). In the context of academic research, it not only structures activities but also reflects the theoretical and philosophical assumptions that shape the selection of methods and techniques for data collection and analysis (Saunders et al., 2019). Whereas in software development, methodologies serve a broadly similar role in providing structure and consistency; however, a distinction arises in that research methodologies guide the conduct and validation of investigation, while software development methodologies guide the delivery and implementation of functional projects (Sommerville, 2016).

### What are methods?

Methods are the specific tools, procedures, or techniques used within a methodology to achieve particular tasks. While a methodology provides the overarching framework, methods constitute the individual activities that operationalise this framework (Saunders et al., 2019).

In research, methods may include interviews, surveys, experiments, or case studies used to collect and analyse data. In software development, methods can refer to coding practices, testing strategies, version control procedures, or user evaluation techniques (Sommerville, 2016).

Methods are selected based on the goals of the project, the nature of the problem being addressed, and the philosophical stance underpinning the research (Saunders et al., 2019).

### Why use a methodology?

In software development, the use of a defined methodology ensures that projects are approached systematically, reducing the likelihood of errors, improving code quality, and enhancing collaboration among team members (Alsaqqa et al., 2020; Fowler & Scott, 2000). Agile, for example, promotes flexibility and responsiveness to change through iterative development (Alsaqqa et al., 2020), whereas Waterfall emphasises thorough upfront planning and linear, sequential progress through distinct project phases (Fowler & Scott, 2000).

In academic research, methodologies uphold the rigour and integrity of the investigative process (Saunders et al., 2019; Creswell & Creswell, 2018). They provide a transparent and reproducible structure that underpins the credibility, reliability, and ethical validity of research findings (Bryman, 2016; Creswell & Creswell, 2018). A robust methodological framework guides researchers in selecting appropriate data collection and analysis techniques, mitigating bias, and ensuring alignment with established academic standards (Saunders et al., 2019; Bryman, 2016).

In project management, methodologies serve as strategic frameworks for resource allocation, scheduling, risk management, progress monitoring, and quality assurance (Office of Government Commerce (OGC), 2009; Schwaber & Sutherland, 2020). Structured approaches such as PRINCE2 provide governance and control mechanisms, whereas Agile frameworks such as Scrum enable adaptive delivery in dynamic environments (Office of Government Commerce (OGC), 2009; Schwaber & Sutherland, 2020).

Overall, the adoption of an appropriate methodology provides a solid foundation for managing complexity, maintaining accountability, and safeguarding the quality of outcomes whether in technical, academic, and managerial domains.

## Software methodologies

In any software development project, the choice of methodology plays a key role in guiding the structure, pace, and quality of work. This project, aiming to deliver an application for real-time code quality assessment, required careful consideration of available methodologies to ensure that the chosen approach aligned with the goals of flexibility, quality assurance, and iterative improvement.

### Agile

Agile is an iterative and incremental development methodology that structures work into short cycles called sprints. It emphasises adaptability, collaboration, and continuous delivery of functional components (Alsaqqa et al., 2020). Agile promotes frequent reassessment of priorities and rapid feedback loops through regular sprint reviews and retrospectives (Alsaqqa et al., 2020).

### Waterfall

The Waterfall model is a linear, phase-based methodology that divides the software development process into distinct stages: requirements gathering, system design, implementation, testing, deployment, and maintenance (Fowler & Scott, 2000). Each phase is completed in its entirety before the next begins. Documentation plays a central role, with deliverables produced at the end of each phase to guide the subsequent stage.

### Test Driven Development

TDD involves writing tests before implementing the corresponding functionality, which fundamentally shifts the development process compared to Agile or Waterfall. Unlike Agile’s iterative cycles (Alsaqqa et al., 2020) or Waterfall’s sequential phases (Fowler & Scott, 2000), TDD centres on ensuring that every piece of functionality is rigorously defined and verified before it is coded (Beck, 2003). This approach not only guarantees high test coverage but also enforces clean, modular design by requiring developers to think through their code’s purpose and structure upfront (Seref & Tanriover, 2016). However, it can introduce overhead, particularly in projects with tight deadlines or where requirements frequently change, as constant test updates may be needed to accommodate evolving features (Kazman et al., 2020).

### Chosen Methodology

Although Agile methodologies are widely adopted in software development for their flexibility and responsiveness to change, the Waterfall model has been chosen for this project due to its closer alignment with the academic research process. The structured, sequential nature of Waterfall supports the clear demarcation of phases which parallels the structured stages of an academic dissertation, including proposal development, literature review, implementation, and critical reflection (Royce, 1970; Sommerville, 2016).

The Waterfall model is particularly well-suited to projects with well-defined objectives and a fixed timeline, such as a final-year project. Each phase concludes with formal documentation and review, ensuring that progress is systematically recorded and that key deliverables are completed before moving on to subsequent stages. This reduces the risk of scope creep within the limited timeframe available for this project (Fowler & Scott, 2000; Adenowo & Adenowo, n.d.; Sommerville, 2016).

While the Waterfall approach does limit opportunities for late-stage changes and emergent features, this constraint is manageable within the context of a tightly scoped academic project. Key functionalities have already been identified during the planning stage, and validation will be conducted using a predefined set of representative code snippets. This enables thorough planning and systematic evaluation, reducing the uncertainty that Agile methods are designed to address.

## Research Onion

This research adopts the research onion framework proposed by Saunders et al (Saunders et al., 2019). A diagram of a scientific method

AI-generated content may be incorrect.

Figure : Research Onion (Saunders et al., 2019)

It comprises six layers, encompassing: philosophies, approaches, strategies, choices, time horizons, and techniques and procedures (Saunders et al., 2019).

Each of these layers was examined sequentially to determine the most appropriate methodological choices.

Other significant factors that can influence methodological choice are ontology, epistemology, and axiology. Saunders et al. define ontology as "the researcher’s view of the nature of reality or being," epistemology as "the researcher’s view regarding what constitutes acceptable knowledge," and axiology as "the researcher’s view of values in research." (Saunders et al., 2019). Although these philosophical considerations underpin methodological decisions, they are positioned outside of the research onion model itself (Melnikovas, 2018).

### Philosophy

When selecting a research philosophy, it is important to recognise that no single philosophical position is inherently superior (Saunders et al., 2019; Phair & Warre, 2021). Instead, the most appropriate choice depends upon the nature of the research problem and the objectives of the study (Thesismind, 2019).

Figure 2 presents a comparison of five research philosophical positions (Saunders et al., 2019).A close-up of a book

AI-generated content may be incorrect.

Figure : Comparison of five research philosophical positions in business and management research (Saunders et al., 2019)

Although originally developed for management research, the distinctions outlined in Figure 2 are sufficiently broad to be applicable within computing research (TechBullion, 2024). They provide a useful basis for selecting the appropriate research philosophy for this project.

In this context, pragmatism was selected as the most appropriate philosophical stance. Pragmatism emphasises the practical application of research, prioritising outcomes that are useful and actionable over the pursuit of absolute truths or solely subjective interpretations (Saunders et al., 2019; Phair & Warre, 2021). In the context of developing a code quality evaluation tool, pragmatism offers the necessary flexibility to iteratively refine methods and techniques based on their effectiveness in meeting the project's goals.

Rather than adhering strictly to positivism, which asserts the existence of a single objective reality, or interpretivism, which considers reality to be socially constructed and subjective, pragmatism adopts a flexible, outcome-focused perspective (Saunders et al., 2019; TechBullion, 2024). Within this project, both measurable results, such as the technical effectiveness of the developed tool, and subjective insights, such as readability of code, are considered valuable, depending on their relevance to addressing the research question. Pragmatism allows the researcher to draw upon a range of approaches, selecting methods based on their practical contribution to the research aims rather than their alignment with any singular philosophical tradition (Saunders et al., 2019; Phair & Warre, 2021; TechBullion, 2024).

### Approaches

In addition to selecting a research philosophy, it is important to determine the research approach that best supports the achievement of the project's aims (Newman, 2024; Streefkerk, 2019). Saunders et al. (Saunders et al., 2019) outline three primary approaches: deductive, inductive, and abductive reasoning (Creswell & Creswell, 2018; Bryman, 2016). Figure 4 provides a comparison of these approaches and their typical applications.

A table of information

AI-generated content may be incorrect.

Figure : Deduction, induction and abduction: from reason to research (Saunders et al., 2019)

For this project, a deductive approach has been adopted. Deductive research is characterised by the application of existing theories or best practices to a specific context, followed by the testing or evaluation of these theories through empirical investigation (Saunders et al., 2019; Creswell & Creswell, 2018; Streefkerk, 2019). Such an approach is particularly appropriate for a project focused on the development of a practical tool, as it enables the grounding of the artefact in established standards and the systematic evaluation of its effectiveness (Bryman, 2016).

The research begins with a theoretical foundation drawn from recognised principles within the field of software engineering. The developed tool is subsequently evaluated to assess the extent to which it supports or reinforces these principles. This logical progression from theory to practice provides a clear and systematic structure, ensuring that conclusions are derived directly from empirical testing against real-world examples (Creswell & Creswell, 2018).

Thus, the deductive approach provides a coherent pathway for validating the tool’s effectiveness, supporting the overall pragmatic philosophy underpinning the research (Saunders et al., 2019; Newman, 2024).

### Strategies

Saunders et al. outline a range of strategies available to researchers, including experiments, surveys, case studies, action research, grounded theory, ethnography, and archival research (Saunders et al., 2019).

A hybrid approach combining action research and case study strategies has been selected for this project. Action research is suited to contexts where the researcher actively develops an artefact that addresses a practical problem while contributing to academic knowledge. In this case, the creation of a code quality evaluation tool forms an integral part of the research process, making action research a natural fit.

For this project, a hybrid approach incorporating both action research and case study strategies has been adopted. Action research is particularly suited to contexts where the researcher is actively involved in developing an artefact to address a practical problem, while simultaneously contributing to academic knowledge (George, 2024).

In the realm of software engineering, action research has been effectively employed to explore the applicability and usability of methods within industrial settings, enabling the development of tools and techniques in collaboration with practitioners (Dittrich et al., 2024). In this project, the creation of a code quality evaluation tool is central to the research process, making action research an appropriate strategy.​

Elements of the case study strategy are also incorporated. Case study research emphasises detailed, contextual analysis of specific instances or examples, allowing for a comprehensive understanding of complex phenomena within their real-life settings (Yin, 2018). In this project, controlled code snippets, designed to simulate realistic development scenarios, serve as the cases through which the tool's effectiveness is evaluated. By analysing how the tool identifies quality issues within these examples, the research can draw detailed conclusions about its practical utility and limitations.

### Choices

The methodological choice defines how data collection and analysis activities are organised within the research. According to Saunders et al. (Saunders et al., 2019), the 'choices' layer of the research onion encompasses three primary options: mono method, mixed methods, and multi-method approaches. A mono method approach involves the use of a single data collection technique and corresponding analysis procedure, either quantitative or qualitative. This approach is particularly suitable when the research aims are narrowly focused, and the data required is homogeneous in nature. Creswell and Creswell (Creswell & Creswell, 2018) support this by stating that a mono method design is appropriate when the research problem is best addressed by a single data type.

In contrast, a mixed methods approach combines both quantitative and qualitative data collection and analysis techniques within a single study, allowing for a more comprehensive exploration of the research problem (Creswell & Plano Clark, 2011). A multi-method approach also employs multiple data collection techniques but confines itself to a single analytical perspective, either quantitative or qualitative (Bryman, 2016).

For this project, a mono method approach has been selected. The primary data collection will consist of controlled technical testing of the developed application. Code snippets will be evaluated using the tool, and the outputs will be assessed against established code quality standards and design patterns. As the project focuses on objective technical evaluation rather than gathering subjective opinions, the mono method approach is both appropriate and effective for supporting the research aims.

### Time Horizons

The concept of a time horizon refers to the timeframe over which research activities are conducted. According to Saunders et al. (Saunders et al., 2019), research may adopt either a cross-sectional or a longitudinal time horizon. A cross-sectional study captures data at a single point in time or over a short, defined period, whereas a longitudinal study involves repeated observations over an extended duration to identify trends or changes.

Cross-sectional studies are particularly useful for assessing the prevalence of phenomena or for identifying associations between variables at a specific point in time. They are often employed in fields such as psychology, social sciences, and public health to provide a snapshot of a population's characteristics or behaviours (Institute for Work & Health (IWH), 2015).However, they are limited in their ability to establish causality due to the simultaneous measurement of variables (Stony Brook Medicine, 2010).

Longitudinal studies, on the other hand, involve collecting data from the same subjects repeatedly over a period of time. This design allows researchers to detect developments or changes in the characteristics of the target population at both the group and the individual level (IWH, n.d.). Longitudinal studies are particularly valuable for understanding long-term effects and causal relationships but require more resources and time to conduct (Verywell Mind, 2023).

Given the structure of this final-year project, which involves the development and evaluation of a software tool within a fixed academic timeframe, a cross-sectional time horizon has been adopted. All data collection and testing activities will be completed within a defined period during the academic year. As the project does not require long-term observation or repeated measurements across time, the cross-sectional approach is both appropriate and aligned with the scope of the research.

### Techniques

The techniques used for data collection in this project are based on structured, technical evaluation rather than qualitative interviews. Data will be gathered through controlled testing of the developed code quality evaluation tool, with a focus is on objective, replicable assessments (Graphite, 2024). Specifically, a series of carefully selected code snippets will be submitted to the tool, and the outputs will be compared against established coding standards and best practices to assess performance and accuracy (Graphite, 2024).

In addition, unit testing frameworks will be used to verify that the tool behaves as expected under different input conditions (GeeksforGeeks, 2025b). These techniques allow for consistent, replicable assessment of the system’s functionality, supporting a rigorous evaluation of its effectiveness.

This technical, artefact-focused approach aligns with the pragmatic philosophy and deductive methodology adopted in the wider research design.

## Project Management Methods

I plan on using git as a version control technology, and GitHub as a hosting platform for my codebase

# Domain Research

## Design patterns

Design patterns represent established, reusable solutions to recurring challenges in software development. Acting as architectural blueprints, they provide a standardised framework for addressing complex design issues efficiently (Refactoring Guru, 2025; GeeksforGeeks, 2025a). Their application promotes the development of robust, maintainable, and scalable software systems (GeeksforGeeks, 2025a; Naumov, 2020). Beyond solving technical problems, design patterns also encourage adherence to best practices, fostering code that is functional, efficient, and readily comprehensible (Shehzad, 2023).

Conforming to design patterns not only enhances the quality of individual codebases but also promotes broader consistency across teams. Patterns provide modular, tested solutions that streamline development workflows, minimise common pitfalls, and reinforce a culture of quality (GeeksforGeeks, 2025a; Gamma et al., 1994). Their systematic application contributes to improved maintainability, scalability, and readability, ensuring that both individual projects and team collaborations benefit from a unified and effective approach to software development (Metridev, 2024; GeeksforGeeks, 2025a).

### Standardisation Across Teams

Design patterns play an essential role in team-based development by offering a shared framework and vocabulary (GeeksforGeeks, 2025a; Gamma et al., 1994). By adhering to common patterns, teams ensure a unified approach to coding, simplifying collaboration and reducing the learning curve for new team members (Shehzad, 2023; Metridev, 2024). A consistent structure not only enhances project maintainability but also fosters clear communication regarding system design and implementation strategies (Naumov, 2020; GeeksforGeeks, 2025a).

### Improved Code Readability and Collaboration

The adoption of recognised design patterns directly improves code readability (Gamma et al., 1994; Refactoring Guru, 2014). Developers and reviewers encountering familiar structures can interpret functionality more quickly and confidently, reducing ambiguity and misinterpretations (Shehzad, 2023; Metridev, 2024). Additionally, this clarity simplifies debugging, streamlines maintenance, and facilitates smoother collaboration, especially within large or distributed teams (GeeksforGeeks, 2025a; Naumov, 2020).

### Enhanced Scalability and Extensibility

Design patterns naturally support the principles of modularity and scalability (Gamma et al., 1994; GeeksforGeeks, 2025a). Well-structured components can be extended or modified without extensive refactoring, ensuring that the system can evolve alongside changing requirements (Metridev, 2024; Shehzad, 2023). This flexibility enables developers to add new features or optimise existing ones with minimal disruption, preserving the system's stability and performance over time(Naumov, 2020; GeeksforGeeks, 2025a).

## Coding best practices

Good coding practices refer to a set of guidelines designed to ensure that software remains maintainable, efficient, and comprehensible (Amoah, 2023). These practices are instrumental in reducing technical debt and long-term development costs while simultaneously enhancing code quality and team productivity (Seref & Tanriover, 2016; Sonarsource, 2025). At the core of these practices is the philosophy of writing code that is not only functional but also easy to understand, often described as "clean code" (Amoah, 2023; Sonarsource, 2025). The following sections explore several key principles that contribute to this goal, including consistent naming conventions, the avoidance of code duplication, self-documenting structures, and adherence to the SOLID principles. Collectively, these practices promote the development of scalable, maintainable, and industry-aligned software (Shehzad, 2023; Refactoring Guru, 2014).

### Adherence to Consistent Naming Conventions

Using meaningful and descriptive names for variables, methods, and classes is fundamental to maintaining code clarity (Amoah, 2023; Refactoring Guru, 2014). Developers should follow language-specific conventions to ensure alignment with community standards, which improves collaboration and understanding among team members (Shehzad, 2023). By avoiding ambiguous or cryptic naming, code readability is significantly enhanced, making it easier for others to work with and extend the codebase (Metridev, 2024).

### Avoiding Code Duplication (DRY principle)

The DRY (Don’t Repeat Yourself) principle focuses on minimising repetition in code by centralising logic (Metridev, 2024; GeeksforGeeks, 2025a). This approach simplifies updates and reduces the risk of bugs by eliminating redundant code blocks. For instance, repeated tasks can be encapsulated within functions or methods, while shared modules can handle common functionality. This not only simplifies maintenance but also promotes a more streamlined and efficient codebase (Amoah, 2023).

### Ensuring Code is Self-documenting.

Self-documenting code is written in a way that its purpose is evident from its structure and clarity (Refactoring Guru, 2014; Amoah, 2023). Developers can achieve this by employing clear logic flows and avoiding overly complex implementations (Shehzad, 2023). Expressive function and method names help convey the intent of the code, while inline comments should be used sparingly for additional clarification when necessary. By promoting readability and reducing reliance on external documentation, self-documenting code facilitates collaboration and maintenance, ensuring that team members can easily understand and build upon the code (Metridev, 2024).

### SOLID Principles

The SOLID principles represent a collection of five key design guidelines that underpin effective object-oriented software development. These principles are widely regarded as fundamental to creating modular, maintainable, and scalable software architectures (GeeksforGeeks, 2025a; Martin, 2003). By adhering to the SOLID principles, developers can significantly reduce system complexity and enhance overall software flexibility (King, 2015; Harm, 2018a).

#### Single Responsibility Principle (SRP)

The Single Responsibility Principle states that each class in a software system should have a single responsibility and, therefore, only one reason to change. This principle ensures that responsibilities remain isolated within distinct classes, simplifying maintenance and minimising the risk of unintended side effects caused by changes in the codebase (Harm, 2018c; Martin, 2003). By adhering to SRP, developers can achieve a more focused and manageable software structure (GeeksforGeeks, 2025a).

#### Open/Closed Principle (OCP)

The Open/Closed Principle asserts that software entities such as classes, modules, and functions should be open for extension but closed for modification. This means that new functionality can be added to existing components without altering their core structure (Martin, 2003). By following this principle, developers can enhance system extensibility while preserving existing functionality, reducing the need for invasive changes when implementing new features (King, 2015; Harm, 2018b).

#### Liskov Substitution Principle (LSP)

The Liskov Substitution Principle requires that objects of a superclass should be replaceable with objects of its subclasses without affecting the correctness of the program. This principle promotes the consistent behaviour of related components and encourages the use of polymorphism (Hombergs, 2020). By adhering to LSP, developers can maximise code reuse and ensure seamless integration of subclass objects into the broader system (Martin, 2003; GeeksforGeeks, 2025a).

#### Interface Segregation Principle (ISP)

The Interface Segregation Principle emphasises that clients should not be forced to depend on interfaces they do not use. By designing focused and specific interfaces, developers can improve cohesion and simplify the implementation and maintenance of system components (Harm, 2018a; Martin, 2003). This principle helps ensure that interfaces remain relevant and efficient for their intended purpose, reducing unnecessary dependencies (King, 2015).

#### Dependency Inversion Principle (DIP)

The Dependency Inversion Principle dictates that high-level modules should depend on abstractions rather than low-level modules. This decoupling of system components enhances flexibility and supports a more modular design (Martin, 2003; GeeksforGeeks, 2025a). By adhering to DIP, developers can substitute dependencies more easily, thereby improving testability and adaptability in software systems (King, 2015).

## Error Handling

Error handling is a critical aspect of software development, aimed at managing exceptions effectively while avoiding fail-silent systems (Shah et al., 2022). Its importance lies in ensuring the application remains functional, protects its integrity, and supports developers in debugging and maintenance through meaningful error reporting (Ray et al., 2016). Proper error handling not only prevents application crashes, enhancing user experience, but also addresses potential issues promptly to maintain system reliability (Grottke & Trivedi, 2008; Zhang et al., 2022).

### Use Specific Exceptions

One of the key practices in effective error handling is the use of specific exceptions instead of generic ones. Generic exceptions often obscure the root cause of problems, making it harder to implement effective remedies (Shah et al., 2022). Developers should therefore prioritise the use of precise, language-specific exception types that accurately reflect the nature of faults. This strategy not only improves the readability of the code but also sharpens the focus during debugging and maintenance (Li et al., 2019). Categorising errors meaningfully allows the development of more nuanced and reliable recovery strategies, ultimately enhancing the resilience of the application (Zhang et al., 2022).

### Centralised Error Logging

Beyond choosing appropriate exceptions, maintaining a detailed record of errors is key to building sustainable software systems (Grottke & Trivedi, 2008). Centralised error logging consolidates error reports in a structured repository, capturing critical information such as timestamps, stack traces, and contextual data (Christakis & Spinellis, 2020; Fu et al., 2014). By maintaining a central repository of error logs, developers can identify patterns, diagnose recurring problems, and refine their debugging processes (Shah et al., 2022). This approach not only supports efficient resolution of errors but also contributes to long-term system reliability and maintainability (Zhang et al., 2022).

### Graceful Degradation

Even with comprehensive error handling and logging, systems must be designed to accommodate failure without collapsing entirely. Graceful degradation ensures that, when errors occur, the application continues to operate in a reduced but functional state (Zhang et al., 2022). Instead of failing silently or crashing, the system informs users through clear, constructive messages, guiding them on possible next steps. This is a common anti-pattern observed in large-scale enterprise systems (Ray et al., 2016; Cabral & Marques, 2005). This not only protects the user experience but also demonstrates the system's robustness and consideration for end-user needs. By planning for partial failure scenarios, developers reinforce trust and usability, even under less-than-ideal conditions (Shah et al., 2022).

### Fail-Fast vs. Fail-Safe

When designing error-handling strategies, it is essential to understand the trade-offs between fail-fast and fail-safe approaches. Both methods offer distinct advantages and are suited to different scenarios depending on the criticality of the system and user experience requirements (Zhang et al., 2022). The fail-fast approach focuses on early error detection and halting execution to prevent further issues (Zheng et al., 2011), while the fail-safe approach aims to recover from errors and maintain a baseline of functionality (Li et al., 2019). The following table contrasts these two strategies:

Table : Comparison of Fail-Fast and Fail-Safe Error-Handling Strategies

|  |  |  |
| --- | --- | --- |
| Aspect | Fail-Fast | Fail-Safe |
| Objective | Detect errors early and stop execution immediately. | Recover from errors and maintain basic functionality. |
| Use Case | Systems where precision and preventing damage are critical (e.g., financial systems). | Systems prioritising uninterrupted operation (e.g., user-facing applications). |
| Response to Errors | Stops execution at the first sign of an issue. | Attempts to handle the error and continue functioning. |
| Advantages | Prevents error propagation and simplifies debugging. | Improves resilience and user experience. |
| Disadvantages | Can disrupt user experience and halt system functionality entirely. | May mask underlying issues or lead to degraded performance. |

By combining aspects of both approaches, systems can be designed to balance error prevention and recovery, aligning with specific project requirements and user expectations (Grottke & Trivedi, 2008).

### Avoiding Fail-Silent Systems:

Fail-silent systems, which fail without providing feedback or clear indicators of the issue, can lead to significant challenges in debugging and user experience (Grottke & Trivedi, 2008; Jin et al., 2012). Studies on production cloud environments have shown that a significant number of real-world failures are initially silent, only surfacing after user complaints or prolonged misbehaviour (Jin et al., 2012). Such faults not only degrade system reliability but also complicate recovery and root cause analysis (Chow et al., 2014). To prevent such scenarios, developers should adopt practices that prioritise early error detection, effective user communication, and transparent exception handling (Shah et al., 2022; Zhang et al., 2022).

#### Detect Errors Early

Validations should be comprehensive and integrated at critical points to prevent errors from propagating through the system. During development, using assertions to identify critical issues ensures problems are identified promptly, reducing the risk of silent failures in production environments (Li et al., 2019).

#### Provide Feedback

When errors occur, it is vital to inform users through clear and concise messages. These messages should explain the issue (e.g., "Invalid email address entered") and provide actionable suggestions for resolution. Avoiding technical jargon in user-facing messages ensures that users can understand and address the problem without confusion (Ray et al., 2016; Shah et al., 2022). In online systems, this is doubly as important, as silent issues can mask widespread service degradation (Oppenheimer et al., 2003),

#### Avoid Suppressing Exceptions

Silent error suppression, such as empty catch blocks, should be avoided as it obscures issues and hampers debugging (Cabral & Marques, 2005). Every exception should be logged or handled in a manner that maintains transparency. Effective error logging ensures that critical details are captured and provides developers with the necessary information to diagnose and resolve problems efficiently (Shah et al., 2022; Zhang et al., 2022; Christakis & Spinellis, 2020).

### Tools for code quality assurance

In a modern development environment, there are numerous tools that exist to aid developers in maintaining coding standards, identifying issues early in the development lifecycle, and improving the overall readability, maintainability, and functionality of software. These tools can be broadly classified into three categories: linters and formatters, static code analysis tools, and AI-driven code quality tools.

#### Linters and formatters

Linters and formatters are commonly used tools for enforcing consistent coding styles and conventions across a codebase. Linters, such as ESLint for JavaScript (ESLint, 2023) or Laravel pint for Laravel PHP projects (Laravel, 2024), analyse code to identify stylistic inconsistencies, syntax errors, and violations of best practices. Formatters, on the other hand, automatically reformat code according to a predefined style guide, promoting uniformity across development teams.

The primary benefit of linters and formatters lies in their ability to reduce cognitive load for developers. By standardising code appearance, they enable faster code comprehension and review. Additionally, automated formatting tools minimise the time spent on trivial styling debates, allowing development teams to focus on substantive aspects of design and functionality. Integrating these tools into the development pipeline, often as part of pre-commit hooks or CI/CD workflows, ensures that code quality checks are automated and consistently applied (Rigby & Storey, 2011).

#### Static code analysis tools

Static code analysis tools operate by examining source code without executing it, providing insights into potential bugs, security vulnerabilities, and code smells. Examples include SonarQube and Codacy, which support multiple programming languages and can be integrated directly into CI/CD pipelines (SonarSource, 2024; Codacy, 2024).

These tools evaluate metrics such as cyclomatic complexity, code duplication, and test coverage, offering developers a detailed breakdown of their code’s structural integrity. By catching defects early in the development cycle, static analysis tools help reduce the cost of fixing bugs and contribute to more robust software (Beller et al., 2016).

These tools can additionally integrate with development environments, allowing for real-time feedback while working. Additionally, the tools in question can be customised extensively, allowing for better alignment with team-specific coding standards, further enhancing their utility (Storey et al., 2017).

The DPVIA (Design Pattern Violations Identification and Assessment) tool exemplifies another significant advancement within static code analysis, utilising Natural Language Processing (NLP) to identify deviations from established design patterns (Ali et al., 2022). Unlike traditional tools that focus primarily on syntactical and structural aspects, DPVIA examines code at a deeper semantic level, evaluating adherence to architectural and design pattern standards. By detecting nuanced deviations from prescribed patterns such as those described in the SOLID principles, DPVIA offers targeted, actionable feedback on structural quality, thereby aiding developers in maintaining high architectural standards and reducing technical debt (Ali et al., 2022; Feathers, 2004).

#### AI-Driven tools

With the advancements in artificial intelligence over the past several years, tools have emerged that can offer intelligent, context-aware support for code quality assurance. Tools such as GitHub Copilot (GitHub, 2024) and Sourcery are capable of more in-depth insights than older methods, like the prior mentioned linters, utilising large-scale pattern recognition to provide real-time suggestions (Sourcery AI, 2024).

GitHub Copilot, as one example, generates code completions and recommendations by leveraging large language models trained on the public repositories of GitHub standard users (GitHub, 2024; Liu et al., 2024). While this aids in rapid prototyping, and can boost developer productivity, it also introduces concerns regarding the quality and security of the generated code. With using user data for training data, the producted code may lack adherence to established design patterns or introduce subtle bugs that are difficult to detect through traditional analysis (Yetiştiren et al., 2023).

Sourcery takes a slightly different approach by focusing specifically on Python code refactoring, offering improvements that align with best practices in code efficiency and readability (Sourcery AI, 2024). Unlike Copilot, which serves a more general-purpose function, Sourcery operates within a more narrowly defined domain, enhancing its reliability for targeted improvements.

Despite their promise, AI-driven tools should be seen as augmentative rather than authoritative. While proficient at producing boilerplate code and recommending commonly adopted patterns, they still require human oversight to ensure compliance with domain-specific requirements and adherence to architectural conventions. As these tools continue to evolve, their integration with traditional static analysis methods may provide a more holistic approach to automated code review (Liu et al., 2024; Yetiştiren et al., 2023).

### Challenges in Enforcing Standards

Despite a range of tools being available for a modern workflow, the consistent enforcement of coding standards across teams and codebases remains a point of focus for many teams. This is due to a mix of human factors, legacy constraints and technological limitations (Seref & Tanriover, 2016; Storey et al., 2017). By understanding these challenges, it is possible to plan more effective strategies to ensure quality and maintainability.

#### Varied Team Skill Levels

One of the more frequent challenges towards maintaining code standards arises from the inherent disparities that will be present between team members experience. Junior developers may lack familiarity with design principles and best practices, while seasoned developers might default to personal conventions that diverge from agreed standards (Bosu & Carver, 2013). This variability can result in code that is difficult to maintain and scale.

Discussions on developer forums such as Stack Exchange reflect a consensus that mentorship and peer review are essential for managing skill disparities in teams, with senior developers guiding junior colleagues to ensure consistent standards (Stack Exchange users, 2011), helping them to build good habits that promote best practice (Rigby & Storey, 2011; Storey et al., 2017).

#### Legacy Systems

Another significant barrier to standardisation is the presence of legacy systems. These systems often comprise outdated, monolithic codebases that were developed without adherence to modern architectural principles such as the SOLID or DRY guidelines, making integration with contemporary tools difficult (Feathers, 2004). The process of refactoring legacy code to align with current standards is resource-intensive, frequently constrained by time, budget, or the potential risk of introducing new faults (Seref & Tanriover, 2016).

However, making incremental improvements over time, such as modularising critical components or introducing automated tests, can gradually improve maintainability and conformance (Kazman et al., 2021).

#### Tooling Limitations

While modern development tools have significantly improved the enforcement of code standards, limitations do persist. Many tools are language or framework specific and can lack full support for custom project configurations, often lacking full support for custom project configurations or domain-specific rules (Ayewah et al., 2008). Static analysis tools may also struggle with dynamic language features, or context-dependent logic, leading to overlooked issues (Beller et al., 2016).

Development tools also often require tuning and updating to remain effective for use with evolving codebases. Without this, the output may become unaligned with current development priorities and be disregarded as a result (Zampetti et al., 2017).

# Evaluation of domain research

## Summary of findings

The domain research conducted has revealed the significant challenges that developers face in maintaining high standards of code quality, particularly in environments augmented by AI-assisted coding tools. While there is broad consensus regarding overarching good practices, the finer domain-specific adjustments required for different use cases vary significantly. While there are tools that exist to help with the endeavour, such as linters, static code analysers, and AI-driven assistants, the research indicates that these solutions often operate in isolation, with specific languages or frameworks, targeting specific issues rather than offering comprehensive, language agnostic, context-aware feedback. Additionally, while the automated tools can aid in achieving conformity, careful configuration and integration within workflows are necessary.

## Aim of the solution

Based on these findings, the proposed solution aims to provide a development assistant capable of analysing user-submitted code for adherence to design patterns, aligning with the recommendation for best practices and coding standards. The tools primary focus is to deliver feedback on if the submitted code meets the criteria established by design patterns, while keeping these criteria flexible, so that they can bend to the needs of any domain-specific adjustments, rather than break as soon as the solution is used under non-standard or specialised conditions. In doing so, it addresses the gap identified in the domain research, for a language-agnostic design pattern focused validation tool.

## Requirements Inferred from Domain Research

Building upon the domain research findings and the subsequent evaluation, several requirements have been identified to guide the selection of appropriate technologies and the overall system design. These requirements are directly informed by the challenges and opportunities outlined in the domain research, ensuring that the proposed solution meaningfully addresses gaps in current practices and tooling. These requirements have been sorted in order of necessity using the MoSCoW approach for a clear distinction between essential, desirable, and non-essential features.

Table : A MoSCoW analysis of project priorities

|  |  |  |
| --- | --- | --- |
| Must have | Critical requirements: without these, the project is considered a failure. | * Language-agnostic analysis * Design pattern conformity analysis * Flexible rule application |
| Should have | Important requirements, but not vital for the system to function. If omitted, there may be minor issues or workarounds needed. | * Near real-time performance |
| Could have | Desirable features that would enhance the solution but are not necessary. | * Context-aware feedback * Integration with existing development tools |
| Won’t have | Features explicitly agreed not to be included in this development cycle. They may be considered for future versions. | * Visually pleasing user interface |

### Language-Agnostic Analysis

Given the wide range of programming languages used across development teams, the solution must operate independently of any single language or framework. This requirement stems from the limitations of existing tools, which are often tailored to specific languages and thus lack flexibility in broader development environments.

### Design Pattern Conformity Analysis

A core finding of the domain research was the importance of design patterns in ensuring scalable, maintainable software. Therefore, the system must assess code not only for surface-level stylistic issues but also for deeper structural conformity to established design patterns, such as those described in the SOLID principles.

### Flexible Rule Application

To accommodate the variability observed in domain-specific coding standards, the system must allow rulesets to be customised or weighted differently depending on context. This flexibility ensures that the tool remains useful across diverse project types without rigid enforcement that could hinder specialised workflows.

### Real-Time or Near Real-Time Performance

Reflecting the need for feedback during active development, rather than post-hoc code reviews alone, the system should be capable of providing real-time or near real-time feedback without introducing significant workflow disruptions, however having some delays is acceptable, provided that they are minimised as best as possible.

### Context-Aware Feedback

The research indicated that fragmented or isolated feedback, lacking awareness of broader codebase structure, limits the effectiveness of analysis. As such, a consideration for the solution providing context aware feedback, that weighs context relationships between classes, modules, and functions is a desirable goal, however as it is potentially difficult to implement this shall remain as a “could have” stretch goal.

### Integration with Existing Development Tools

Recognising the importance of continuous integration and streamlined development workflows, a consideration should be given to the integration with existing development workflows, such as CICD pipelines, but given the added complexity that this will add to the development of a solution, this will also have to be listed as a stretch goal.

### Usability and Accessibility

While enhancing the developer experience is the objective of the project, it does not inherently necessitate the inclusion of a comprehensive user interface. Many developer-focused tools operate effectively through command-line interfaces or utilise minimalistic designs prioritising functionality over user comfort. In this project, the decision has been made to deprioritise the development of an advanced user interface, given the limited development resources available. Efforts will instead be concentrated on implementing adding more functionality to the solution, rather than visual improvements

# Technology Research

## Initial Considerations of the Proposed Solution

Based on the requirements identified in Sections 4.3.1 to 4.3.7, the following metrics have been established to guide technology selection:

* Language-agnostic capability: Ability to handle multiple programming languages without dependency on specific syntax trees or frameworks.
* Support for design pattern analysis: Capacity to assess structural and architectural conformity beyond basic syntax or stylistic checks.
* Flexible rule application: Ability to configure or prioritise validation rules to accommodate domain-specific standards.
* Real-time or near real-time performance: Ability to provide feedback quickly enough to integrate seamlessly into development workflows.
* Context-aware feedback: Ability to use a multiple classes and functions included in context analysis for wider analysis
* Ease of workflow integration: Compatibility with existing tools such as version control systems and CI/CD pipelines
* Maintainability and stability: Long-term viability, documentation support, and ecosystem maturity of the chosen technologies.

With the priorities of these requirements described in Table 2, the analysis of technologies will be weighted according to their degree of requirement for the solution, with the “must-have” requirements being the most critical, and prioritisation decreasing as research moves through the MoSCoW analysis of Table 2.

## Evaluating Large Language Models for Code Analysis

To achieve the language-agnostic capabilities that have been identified as a key metric, it is valuable to consider following a similar method from the source in the literature with the closest parallels to this project, the “Software Design Pattern Violations Identification and Assessment Tool” (DPVIA), that utilises Natural Language Processing.

Following this example, the following section is a critical evaluation of currently available large language models and their efficacy as it relates to code analysis.

### Overview Comparison of Current Models

In Table 3, below, you can see an overview comparison between three of the most popular LLM models currently available on the public market, ChatGPT-4, Claude and Gemini.

Table : Comparison of the features between current models

|  |  |  |  |
| --- | --- | --- | --- |
| Feature | GPT-4 | Claude | Gemini |
| Context Retention | Handles up to 128K tokens (GPT-4 Turbo) in the latest models. | Extensive memory with up to 200K tokens in Claude 2.1, designed for long-form recall. | 1 million tokens in the Gemini 1.5 model, allowing for significantly larger context windows. |
| Strengths in Context Handling | Retains logical coherence over long exchanges but may struggle with deep contextual dependency over extended sessions. | Optimised for summarisation and maintaining consistent narrative flow across extensive inputs. | Capable of processing multiple code files simultaneously, beneficial for tasks requiring extensive cross-referencing. |
| Hallucination Risk | Prone to fabricating plausible-sounding but incorrect details, especially in niche or technical domains. | Generally less prone to hallucinations compared to other models due to more cautious response tuning. | Large context size mitigates hallucinations by keeping more previous data within reference, but hallucinations still occur in edge cases. |
| Best Use Cases | General-purpose AI suitable for structured tasks like coding, writing, and problem-solving. | Research-heavy and analytical tasks due to better factual consistency in long-form outputs. | Complex multi-file analysis (e.g., large codebases, legal documents) where retaining vast amounts of data is crucial. |

The most important metric to draw attention to here is the “Context window” available to each model. The context window determines the number of tokens that can be parsed into the model at once, to be factored into analysis. This directly relates to the “design pattern adherence analysis”, and the “context-aware feedback” criteria stated in section 5.1 as without a sufficiently sized context window, holding both the information about the design pattern criteria and the class file it is being compared to will be impossible, and the reliability of the analysis performed will decline. Additionally, if the stretch goal of utilising multiple class files and design patterns simultaneously is to be incorporated, then having the larger context window will be necessary (OpenAI, 2024; Anthropic, 2024; DeepMind, 2024).

In addition to context window size, each model should be assessed on its ability to handle multiple programming languages in a flexible and framework-independent manner, in accordance with the language-agnostic capability requirement stated in section 5.1, without requiring custom syntax tree generation or language-specific parsers.

GPT-4 offers broad multi-language support and performs reliably for structured tasks such as code generation, summarisation, and problem-solving. Its context window of up to 128K tokens enables it to handle moderately sized, single-language or mixed-language inputs. However, GPT-4 tends to prioritise syntactic correctness over architectural depth, relying on token-level prediction rather than maintaining strict structural consistency across languages. While it is effective for smaller tasks and general coding assistance, its performance may degrade when faced with complex, multi-file, multi-language architectural analysis(OpenAI, 2024).

Claude 2.1 improves upon GPT-4’s consistency across larger inputs, with an expanded 200K token context window. Its strength lies in summarisation and maintaining narrative flow, which allows it to track logical coherence across longer, mixed-language inputs more effectively. Nevertheless, Claude’s cautious response tuning, designed to minimise hallucination, occasionally limits its ability to deeply interpret structural relationships between code components, especially where idiomatic or domain-specific language usage diverges (Anthropic, 2024).

Gemini 1.5 exhibits the most advanced language-agnostic capability among the models evaluated. Its unprecedented 1 million token context window allows for the simultaneous parsing and cross-referencing of multiple classes, functions, and design patterns, even across different programming languages. Gemini’s design prioritises structural comprehension over surface-level syntax, enabling more accurate detection of modularity, inheritance structures, and pattern conformity without reliance on language-specific ASTs (Abstract Syntax Trees). This makes it particularly suited for the project’s objectives, where a flexible and scalable architecture analysis is required across varied codebases (DeepMind, 2024).

In summary, while GPT-4 and Claude demonstrate competent language handling for smaller or more homogeneous inputs, Gemini’s ability to maintain structural awareness across extensive, multi-language contexts establishes it as the most capable model for delivering reliable, language-agnostic analysis.

### Hallucination risks

Another one of the criteria identified in section 5.1, is the “maintainability and stability” of the chosen technologies. In the context of using LLM’s for code analysis, “hallucination”, where models generate plausible but factually incorrect information is the primary concern as it relates to the trustworthiness of the analysis output, and the stability of the technology at hand.

Hallucinations can lead to erroneous feedback, misidentification of design pattern violations, or incorrect structural advice, undermining developer confidence and requiring additional verification effort. In a production environment, this instability increases maintenance burdens, as errors introduced by the analysis tool must be caught, corrected, and documented over time.

Evaluating the three models, GPT-4 exhibits a higher hallucination rate, particularly when dealing with niche or domain-specific code (OpenAI, 2024). While manageable for general-purpose coding assistance, this instability makes it a less viable choice for a maintainability-focused tool, especially when factoring the “flexible rule application" criteria that this is being evaluated against.

Claude 2.1 offers improved factual consistency compared to GPT-4 through more cautious response tuning. However, its conservative nature can occasionally limit deep structural analysis, creating gaps in insight that could require frequent human oversight and, consequently, increased maintenance overhead (Anthropic, 2024).

Gemini 1.5, while not immune to hallucination, shows significant advantages due to its expanded context window discussed in 5.2. By retaining more source material during analysis, Gemini reduces the risk of incorrect outputs. Importantly, its structural prioritisation over purely surface-level syntax suggests that hallucinations are less likely to affect architectural evaluations, making it a stronger candidate for long-term maintainability (DeepMind, 2024).

From a stability and documentation perspective, models prone to hallucination introduce difficulties in both updating the system and ensuring reliable documentation of its outputs. A tool based on a less hallucination-prone model like Gemini will be easier to document, validate, and maintain over time, as updates to accommodate model changes or behaviour shifts will be less frequent and more predictable.

In conclusion, while all models introduce some risk of hallucination, Gemini 1.5 offers the best alignment with the project's maintainability and stability requirements. Its ability to handle large, multi-file contexts reduces the likelihood of fragmented or fabricated feedback.

## Backend Development Framework

Given the importance of the requirement for language-agnostic analysis identified in sections 4.7.1 and 5.1, and following the evaluation of existing LLM models in section 5.2 that shows the Gemini 1.5 model as a clear choice for the code analysis, selecting a backend framework that offers robust support for file handling, scalability, and integration with the Gemini 1.5 API was pivotal. Gemini's official support documentation identifies Python, Node.js, Go, and Shell scripting as compatible options. Consequently, the investigation focused on Python and Node.js, assessing their capabilities against the project’s key criteria: maintainability, language flexibility, integration potential, and support for rigorous testing strategies.

### Python

Python is a widely used language in software engineering and data science, favoured for its readability, versatility, and extensive library support (DataCamp, 2025; Flare, n.d.). Among the numerous frameworks available, Flask and FastAPI are two of the most prominent micro-frameworks suited for building RESTful APIs (BetterStack, 2025; FastAPI, 2025a).

Flask is a minimalist web framework that provides core routing functionality and encourages extension through third-party modules. Its simplicity and explicit design make it ideal for small to medium applications where fine-grained control is desirable (Projects, n.d.; Vidhya, 2021). In contrast, FastAPI is a modern framework built on top of Starlette and Pydantic, offering automatic validation, documentation (via Swagger and ReDoc), and high performance through asynchronous request handling (FastAPI, 2025b). FastAPI is particularly suited to applications requiring robust request validation and rapid iteration, making it an ideal candidate for integrating AI-driven analysis endpoints that require structured request data (BetterStack, 2025; FastAPI, 2025b).

### File Handling

As shown in the Gemini documentation, to utilise analysis features on a file using Gemini, you need to send the file with it, meaning that file upload capabilities are a key requirement (Google, 2025). Python, through both Flask and FastAPI, offers efficient mechanisms for file upload, processing, and validation.

In Flask, file handling is managed through the request.files object, which supports multipart form data submissions and integrates seamlessly with utilities such as python-magic for file type detection and Werkzeug’s secure\_filename for sanitising filenames (Projects, n.d.).

FastAPI further extends this with its dependency injection system, allowing developers to explicitly declare file inputs using the UploadFile type. Files are streamed in asynchronously, minimising memory consumption and enhancing performance when handling large inputs (FastAPI, 2025b).

Both frameworks allow for secure temporary storage and sanitisation of uploaded content, and they integrate well with file both system libraries and external APIs (BetterStack, 2025; FastAPI, 2025a).

### Unit Testing and Validation Strategy

Python’s testing ecosystem is mature, with several frameworks supporting Test-Driven Development (TDD), behavioural testing, and code coverage analysis. This section explores the three most pertinent options for this project: unittest, pytest, and nose2.

#### Unittest

unittest is Python’s built-in testing framework, offering a class-based structure similar to JUnit in Java (Python Software Foundation, 2024). It supports test discovery, setup and teardown methods, and test fixtures. Its tight integration with standard Python tooling and IDEs such as PyCharm makes it accessible for developers new to testing (JetBrains, 2024).

Despite its accessibility and reliability, unittest is often critiqued for its verbosity. Assertions must be written using explicit method calls like self.assertEqual() or self.assertTrue(), rather than using native Python expressions, which can reduce test readability and slow down development compared to more concise frameworks such as pytest (GeeksforGeeks, 2025b; Kuhlman, 2020).

#### Pytest

pytest has emerged as the de facto standard for modern Python testing due to its clean syntax, powerful fixture management, and expansive plugin ecosystem, including tools like pytest-cov for generating coverage reports (Krekel, 2024; GeeksforGeeks, 2025b). It allows developers to write concise, readable tests using native assert statements, which improves both test maintainability and developer experience.

Its support for advanced features such as parameterisation and fixture scoping enables the creation of DRY and scalable test suites, making it suitable for complex projects (Kuhlman, 2020). Furthermore, pytest maintains compatibility with existing unittest codebases, offering a straightforward migration path for legacy projects. Given its balance of readability, extensibility, and community support, pytest is the preferred framework for this project.

#### Nose2

nose2 is the official successor to the deprecated nose framework, designed to maintain compatibility with unittest-style test cases while offering automatic test discovery and support for configuration via plugins (Nose2 Developers, 2024). It preserves many of the original nose features while providing a cleaner, modular architecture.

However, despite its functional design, nose2 has a comparatively smaller user base and slower development activity, raising concerns about long-term viability and community support (Kuhlman, 2020). In contrast, pytest offers a broader ecosystem, frequent updates, and extensive plugin availability, making it the preferred choice for projects requiring robust community backing and extensibility.

### Node.js

Node.js is a JavaScript runtime environment built on Chrome’s V8 engine, offering an asynchronous, event-driven architecture that is particularly suited to high-concurrency applications (Tilkov & Vinoski, 2010). Its non-blocking I/O model allows it to handle multiple simultaneous operations efficiently, which makes it ideal for RESTful APIs that must support real-time feedback loops or frequent file uploads, as required by this project.

In the context of code analysis pipelines, Node.js is especially appealing due to its performance characteristics, lightweight footprint, and mature ecosystem of libraries (Tilkov & Vinoski, 2010). Coupled with Express.js, a minimalist and unopinionated web application framework, developers can implement clean, modular REST endpoints with support for routing, middleware integration, and extensibility (Holmes, 2016). Express enables a Model-View-Controller (MVC) architectural pattern, facilitating separation of concerns and maintainability of the backend codebase(Holmes, 2016).

### File Handling

For applications requiring reliable file upload workflows, Node.js provides a powerful ecosystem of middleware libraries. In this project, Multer is employed to manage incoming code files.

Multer acts as middleware for Express routes, supporting multipart/form-data parsing and enabling validation of uploaded files before they reach core logic (Multer, 2025). Developers can define strict rules on file size, MIME type, and storage strategy (in-memory or disk), which are essential considerations when handling untrusted user-submitted code (Express js, 2025).

This capability is crucial in ensuring that malformed or malicious files are filtered early in the processing chain. Furthermore, Multer integrates smoothly into Express’s request lifecycle, allowing validation, sanitisation, and error-handling routines to be consolidated within the middleware stack (Multer, 2025). Its performance characteristics and ease of use make it a dependable solution for scalable file handling in analysis-driven applications.

### Unit Testing and Validation Strategy

Testing is a cornerstone of high-quality software development. For Node.js-based systems, multiple testing frameworks exist that support both unit and integration testing, mocking, and behaviour verification (López et al., 2018). This project evaluates two primary options: the Mocha-Chai-Sinon stack and Jest.

#### Mocha-Chai-Sinon

The Mocha-Chai-Sinon combination is a modular testing architecture frequently used in professional Node.js development (López et al., 2018).

* Mocha provides the test runner and structure, supporting asynchronous testing and nested test suites, which makes it suitable for validating complex workflows such as file uploads and analysis triggers.
* Chai adds an expressive assertion syntax, including BDD-style constructs (e.g., expect and should) that enhance readability and maintainability.
* Sinon introduces test doubles—spies, stubs, and mocks—which allow developers to isolate units and verify interactions with dependencies such as external APIs or file parsers.

This triad enables thorough and flexible testing, particularly in modular applications. Its fine-grained control over mocking and test structuring aligns with the demands of this project, where backend components need to be tested in isolation from the LLM analysis layer (López et al., 2018).

#### Jest

Jest is an all-in-one testing framework developed by Meta, known for its ease of setup and comprehensive feature set. It includes a test runner, assertion library, mocking utilities, and code coverage tools out-of-the-box (Meta, 2023).

Jest’s zero-configuration approach allows developers to rapidly scaffold test environments, making it especially attractive for small teams or rapid prototyping. However, its monolithic design can be limiting when working with modular architectures or when specific mocking patterns are needed. In contrast, Mocha-Chai-Sinon provides more granular control, which may better support this project’s layered validation and analysis system (López et al., 2018).

Nevertheless, Jest remains a viable alternative, especially for test-driven development (TDD) and projects seeking a quicker setup with fewer dependencies. Its active community and integration with CI tools further enhance its appeal (Meta, 2023).

### Evaluation Against Project Requirements

An evaluation of backend frameworks against the project’s functional and non-functional requirements identified in Section 4.7.1 supports the adoption of a Node.js-based architecture.

Language-Agnostic Analysis: Node.js, like Python, is agnostic to the structure or syntax of uploaded source files. It handles files as binary objects without imposing language-specific parsing, ensuring compatibility with the project’s requirement for multi-language support.

Maintainability and Stability: Node.js combined with Express.js and a modular testing framework (Mocha, Chai, Sinon) facilitates a clean separation of concerns. This supports a maintainable and extensible codebase aligned with SOLID principles. The ecosystem also benefits from extensive documentation and community support, contributing to long-term sustainability.

Real-Time Performance: The asynchronous, event-driven architecture of Node.js is particularly well-suited to handling concurrent file uploads and outbound requests to LLM APIs, aligning with the requirement for real-time or near-real-time feedback. Performance under I/O-intensive conditions is further enhanced by middleware such as Multer, allowing for efficient input validation and pre-processing.

While Python offers comparable capabilities via Flask or FastAPI, Node.js exhibits greater synergy with the target architecture’s non-blocking, event-driven paradigm. This renders it a more appropriate choice for a feedback-intensive, user-driven application.

## Automated Analysis and Workflow Integration

One of the stretch goals identified in sections 4.7.1 and 5.1 was for the integration with existing development tools. This likely would take the form of repository analysis, leveraging GitHub's API capabilities to support accessing version-controlled source code, which is a vital component of modern development workflows, especially in relation to continuous integration and deployment (CI/CD) practices (Microsoft, 2025).

The GitHub API offers extensive functionality for interacting with repository metadata, commit histories, and file structures. Integrating this API will enable the application to automatically retrieve source code from user-specified repositories, rather than requiring manual submission of user-submitted code for analysis. This would reduce the wait time on analysis, in line with the “near real-time performance” criteria from section 4.3, reducing the friction between code creation and quality assessment (Microsoft, 2025).

There are two pathways that this could be done by. Manual selection from an integrated repository, and a fully automated webhook triggered automatic feedback system.

### Manual repository analysis

As its initial implementation, the system could allow users to authenticate via OAuth or personal access tokens and select repositories for analysis through a secure interface. Once a repository is selected, the application could extract relevant files, perform the analysis, and deliver feedback. Such an approach aligns with early tooling concepts discussed in open-source initiatives, such as GenAIScript’s Pull Request Reviewer (and Mahendra Chouhan’s PR-Reviewer tool, which support rule-based code inspection and natural language feedback (Chouhan, 2024).

### Webhook triggered feedback

A more advanced integration would involve the use of GitHub webhooks to facilitate real-time, event-driven analysis. Webhooks can be configured to trigger scans upon predefined repository events such as pushes, pull requests, or tag creations. This aligns the tool more closely with CI/CD pipelines, and the integration with existing tools criteria specified earlier (Anon, 2024a; Reed, 2024).

Immediate feedback on commits and pull requests would reinforce best practice adherence at the earliest stages, reducing the risk of accumulating technical debt. This approach is being adopted by an increasing number of projects which seek to generate intelligent feedback before code is merged, sometimes even suggesting improved commit messages through LLMs (Reed, 2024).

Implementing webhook-driven analysis introduces challenges, however, particularly in relation to system scalability and response latency.

The system must handle concurrent webhook events without degrading performance, preserving the "near real-time" criterion. Also, to support more diverse development environments, an even further goal could be to explore integration with other version control platforms such as GitLab and Bitbucket, to enhance the generalisability of the solution.

## Risk Acknowledgements and Limitations

While the proposed system is designed to provide language-agnostic analysis and adaptive rule evaluation, there are several limitations and associated risks that must be acknowledged. Foremost among these is the recognition that full language-agnosticism remains a complex and evolving challenge. Programming languages differ significantly in terms of syntax, semantics, idiomatic usage, and design paradigms. Although the adoption of a large language model with an extensive context window (e.g., Gemini 1.5) enables flexible multi-language support, this capability is constrained by the limitations of training data coverage, model biases, and evolving language-specific conventions. Consequently, some languages or code structures may not be evaluated with consistent accuracy, potentially leading to uneven analytical outcomes.

Additionally, code quality itself is not always an objective measure. While metrics such as cyclomatic complexity or adherence to design patterns offer tangible indicators, elements such as readability and brevity are inherently subjective. Developers may prioritise clarity, while others may favour conciseness or performance. These differences introduce variability in how feedback is perceived and acted upon. Although the system attempts to mitigate this through configurable rule weightings, it cannot fully account for personal or team-level stylistic preferences, which may result in resistance to standardised feedback mechanisms.

To address these limitations, the system has been designed with an emphasis on transparency and flexibility. All analysis results will be accompanied by clear justifications referencing the rule sets applied, allowing developers to trace recommendations to their sources. Furthermore, rule configurations will be adjustable, enabling users to customise the tool’s feedback to better align with project-specific or organisational standards. This ensures the tool remains a supportive assistant rather than an inflexible enforcer, fostering trust and long-term utility within diverse development contexts.

Ultimately, while these risks introduce challenges to the system’s universal applicability and precision, the project embraces them as opportunities for iterative refinement and user-led evolution. By foregrounding adaptability and clear communication, the system aims to maintain relevance and usability despite the inherent complexity of its target domain.

# Design

## Requirements Summary and Design Input

As this report has already identified a complete set of functional and non-functional requirements through domain (Section 3) and technology research (Section 5), the purpose of this phase is to consolidate those findings into formalised design inputs. This mirrors the initial requirements gathering stage of the Waterfall model, where all specifications must be finalised prior to system design (Fowler & Scott, 2000).

The consolidated requirements are as follows:

Functional Requirements

* The system shall allow users to upload source code files or snippets via a file upload mechanism
* The system shall analyse the submitted code for adherence to recognised design patterns
* The system shall return structured, actionable feedback to the user in a readable format

Non-Functional Requirements

* The system shall support analysis across multiple programming languages without reliance on language-specific ASTs (language-agnostic analysis)
* The system shall provide responses in near real-time
* The system shall operate through a lightweight interface (CLI or basic local server) with no dependency on a GUI

Requirements are further prioritised using the MoSCoW framework developed in Section 4.7. This forms the basis for system design decisions in the following stage.

## System Design

This phase translates the requirements into a complete system architecture. The design adheres to modular principles, separating concerns across layers.

The system backend will be implemented using Node.js in conjunction with the Express framework and MVC architecture, enabling the development of RESTful API endpoints responsible for handling file uploads and triggering code analysis. To support secure and efficient management of user-submitted files, the application will integrate with Multer, a middleware solution for processing multipart form data.

The core analysis functionality will be delivered through integration with the Gemini 1.5 API, which enables large language model (LLM)-driven evaluation of the submitted code against established design patterns such as those defined by SOLID principles.

Following analysis, a results parser will interpret the raw output from Gemini, transforming it into a structured format suitable for end-user consumption.

Interfaces are defined using JSON schemas. Some endpoints include:

POST /upload: Accepts user code

POST /analyse: Sends code to Gemini and receives structured feedback

## Implementation

The implementation phase involves translating the system design into a functional application. The following were implemented:

* Set up project structure and version control (Git + GitHub)
* Develop backend API in Node.js with Express
* Implement file upload with Multer
* Connect to Gemini 1.5 and implement prompt for design pattern evaluation
* Parse Gemini response and format feedback

## Testing

The test strategy focused on verifying both functionality and reliability:

* Error Handling Tests: Simulate invalid input types, malformed files, and failed API requests.
* Controlled Scenario Tests: Evaluate output accuracy using known test cases that violate or conform to design patterns (e.g., Singleton, Factory).
* Unit Tests: Cover route handling, file validation, and parser logic.

Test outcomes are discussed in Section 8

* 1. Deployment and Maintenance

The artefact is not intended for public deployment; however, it is packaged for local use. Key points include:

* Local Hosting: Runs via a command-line script or npm start to launch a REST API locally.
* Modular Extensibility: File handling, analysis, and parsing components are abstracted for potential replacement.
* Maintenance Considerations: While no formal maintenance cycle is planned, modularity supports easy patching and expansion.

The system is ready for further development in collaborative or production contexts via containerisation or integration with version control workflows.

# Project testing

Given the scope of this project, testing focused primarily on unit tests and controlled evaluations using class files with predefined expected outcomes.

## Testing Methodology

The testing methodology for this project included a combination of unit tests and validation tests with known class files. These tests were specifically selected to verify that the system’s analysis of code adherence to recognised design patterns and coding standards was accurate and reliable.

## Unit Testing

Unit testing was employed to systematically verify individual components of the system, such as route handling, file validation processes, and parser logic. The chosen testing frameworks were Mocha, Chai, and Sinon, selected for their extensive support for mocking, assertions, and asynchronous testing capabilities. Unit tests were structured following the Arrange-Act-Assert pattern to ensure clarity and maintainability.

Example of unit tests conducted:

* File CRUD handling tests
* Error handling scenarios (e.g., malformed input, failed API requests)
* Parsing functionality for transforming raw API outputs into structured feedback

## Validation Testing

Validation tests were designed using class files that exhibit clear adherence or violations of established design patterns, such as the Singleton or Factory patterns. The expected outcomes were predefined based on authoritative sources, such as design pattern literature and coding best practices outlined in prior research sections.

Examples of validation tests:

* Singleton pattern detection (positive and negative scenarios)
* Factory method implementation adherence
* Web manifest validation

## Test Results and Evaluation

The results from the testing phase demonstrated that the artefact effectively identified both compliance and violations in the supplied test cases. Specific outcomes included:

* Accurate detection of correct and incorrect design pattern adherence
* Correct status codes for route handling
* Correct views being served

## Limitations of Testing

The scope of testing was constrained by project timelines, and the unit tests primarily covered functional logic rather than extensive integration or system-wide tests. Additionally, the reliance on a third-party API (Gemini) introduced an element of unpredictability in test outcomes, complicating the interpretation and consistency of results.

# Critical Reflection

## Project Success

### Artefact Functionality

The artefact successfully fulfilled all "must-have" requirements outlined in Section4.3, including language-agnostic analysis, flexible rule configuration, and functional feedback provision. It demonstrated compatibility with multiple programming languages, notably C# and Java, and also supported structured formats such as web manifests, indicating generalisability in scope that was not explicitly planned for.

Performance testing confirmed that the tool consistently returned results in under 20 seconds. While not immediate, this is within an acceptable range given the absence of a strict definition for "near real-time" and the involvement of large language models (LLMs) in the analysis pipeline.

However, the reliance on LLMs introduced brittleness into the analysis process. Minor changes in prompt structure occasionally produced significantly divergent outputs, some of which were unusable or inconsistently formatted. Additionally, the Gemini API does not return explicit error codes for failed analyses, making fault diagnosis particularly difficult. Although the tool remained stable during edge cases, it sometimes returned ambiguous or inconsistent feedback.

### Usability and Developer Experience

The user interface, while intentionally minimal, was clear enough to support interaction. As noted in Section 5.1, interface design was not a core priority, which is reflected in the final artefact. The current setup requires local hosting via an Express server, introducing friction in environments where developers may already be running services on localhost, resulting in potential port conflicts.

From personal workflow testing, it was evident that while local hosting is viable for individual use, it would prove problematic in collaborative scenarios. Shared access would necessitate centralised deployment or containerisation. Furthermore, the use of per-request API keys for LLM access introduces challenges for group use, as there is no user-specific rate limiting. This may lead to rapid credit depletion and increased operational cost.

The perceived value of feedback is highly dependent on user experience. For junior developers, particularly students learning design patterns, the tool could serve as a formative learning aid. However, this is predicated on the relevance and accuracy of the evaluation criteria. The need for manual updates to these criteria, particularly as requirements evolve, is time-consuming and requires advanced technical understanding. This limits the tool’s long-term usability, both for less experienced users working independently and for senior developers who may find the overhead of maintaining the criteria outweighs the benefit provided.

### Extensibility

The architecture was designed with future extensibility in mind. In particular, the file upload system, initially handled via the Multer middleware, has been modularised in such a way that it could be replaced with alternative file sources, such as integration with Git repositories or direct file system access, as discussed in 4.3.6

In retrospect, research on the application of a dependency injection framework would have further improved the system’s extensibility. By decoupling core components from their specific implementations, this would have also allowed for easier testability. While not inside the current scope, this architectural refinement could be a valuable enhancement in future iterations of the tool.

## Extent of Learning

### Technical Skills

Prior to this project, I had an understanding of Node.js, Express.js, and MVC architecture, though I had not previously used Multer. Through implementation, I found Multer to be straightforward and effective for handling file uploads, and the experience has reduced my hesitancy towards adopting unfamiliar middleware in the future.

The Mocha-Chai-Sinon stack presented a steeper learning curve than Jest, with which I had prior experience. The initial setup was initially complex, but once configured, the flexibility and clarity of the testing was superior to Jest, especially for a more complex project. Moving forward, for complex Node.js applications I will absolutely be viewing it as the preferred testing stack.

Given the widespread integration of LLM’s at the current moment, I expected the experience of incorporating it to be less challenging than it ended up being. While getting a working version of an LLM integrated was easy, I found it unreliable and unpredictable in the context of code analysis. This experience highlighted the importance of prompt design to me and exposed the limitations of LLMs, particularly their brittleness and inconsistency.

### Research and Evaluation

The review of existing literature and available tools revealed that most current solutions are highly specialised, often supporting only a single programming language. When I came to writing the evaluation of the domain research, I saw this as presenting an opportunity to develop a more versatile tool that leverages emerging technologies. While the implementation achieved partial success, the integration of a large language model (LLM) for broad, language-agnostic analysis introduced notable limitations, as discussed in Section 8.1.1, so much so that it is now abundantly clear to me the reason that this opportunity has not been taken by somebody else already.

My understanding of code quality has improved massively, prior to starting this project I had never heard of design patterns before, let alone used them extensively. I was familiar with DRY, but any of the SOLID principles were a foreign concept to me.

### Project Management

The project management aspect of this work was significantly hindered by both my initial misunderstanding of the research process, and my personal challenges with balancing multiple distinct workloads at once. Having never done a dedicated research project before, I was unfamiliar with how a standard research project is structured, which led to sections such as my technology research encroaching on the time allocated for implementation. A considerable amount of project time got stuck on pursuing an unproductive direction, despite consistent efforts by both my supervisor during weekly meetings and my assessor at the Mid-Project Review (MPR) to steer the project back on course.

There was also one large setback of the schedule, being when the Gemini file manager was shutdown, with no forewarning that I was aware of. This was an integral part of the system, and something I had been increasingly building around. This being shutdown meant that I had to revert back to a much earlier version of the artefact, losing weeks’ worth of progress.

## Future enhancements

### GitHub Integration

A stretch goal identified in Section 4.3.6 and expanded upon in Section 5.1 was the integration of Git. While technically feasible, this represented an ambitious undertaking that was ultimately constrained by time. Implementing Git integration would have necessitated a complete replacement of the file manager component. Although the system was architected to support such extensibility, the unexpected deprecation of the Gemini file manager significantly disrupted development timelines, leaving insufficient time for this feature to be revisited.

### Context aware feedback

This feature was included as a stretch goal in the original development plan and initial implementation had commenced. However, progress was entirely lost following the deprecation of the Google file manager, which had been central to storing and managing the required metadata. A revised approach was subsequently scoped, involving the construction of an internal file map to replicate the previously external metadata structure. Nevertheless, due to the project’s revised timeline and limited remaining development capacity, this alternative could not be realised within the available time frame.

### Performance Optimisation

In addition to considerations regarding token consumption and associated costs, the artefact would require enhancements to support multi-user environments. In its current state, all users share a common pool of uploaded files, which introduces the risk of data conflicts. Even with the possible implementation of Git integration, the absence of proper server-side handling for branching mechanisms could result in file collisions and inconsistencies across sessions. Addressing these issues would be essential for deploying the tool in collaborative or production settings.

### Career Influence

This project has had a significant impact on my future career aspirations. Contrary to what might be expected, it did not foster a deeper interest in AI tooling. Instead, it highlighted several critical limitations inherent in current AI development models, particularly the brittleness of prompt-based interaction and the opacity surrounding the design and operation of commercial large language models. As a result, I am now disinclined to pursue work in this area under prevailing industry conditions.

Conversely, the project reaffirmed my interest in developer experience (DX) and software testing. I have long valued the role of comprehensive unit testing and will continue to prioritise it in future projects. The project also introduced me to continuous integration and deployment (CI/CD), which I now intend to explore further. Since adopting Git approximately 18 months ago, I have become increasingly interested in expanding my version control practices to include structured pull request workflows and the use of tools such as PHP Pint for code formatting and quality assurance in Laravel-based applications.

### Confidence and Growth

This project presented considerable emotional and cognitive challenges, and had a significant impact on my confidence. My limited understanding of the structure and expectations of academic research projects, particularly in the early stages, contributed to a sense of uncertainty. By the time of the Mid-Project Review (MPR), the project was not progressing as intended, and I experienced substantial motivational difficulties. Much of the literature I had engaged with up to that point did not offer a clear direction, and I felt constrained by the topic choice made during the proposal stage.

The unexpected deprecation of the Gemini file manager and the consequent loss of a substantial amount of development work further undermined my confidence. At that point, I seriously considered deferring the project and making a new attempt the following academic year. However, consistent encouragement and constructive feedback from my supervisor helped to restore my focus and morale. Reflecting on the overall experience, I now feel significantly more capable of conducting structured research in the future, with appropriate planning.

### Awareness of Industry Practices

Through this project, I have developed a more comprehensive understanding of the practices and considerations involved in producing production-ready software. My exploration of coding standards and CI/CD methodologies has clarified the importance of structured workflows, testing, and deployment processes in professional environments.

Furthermore, I have gained a more realistic perspective on the trade-offs inherent in software development. Operating within strict time constraints necessitated difficult decisions regarding the prioritisation of features and technical debt. This experience has helped me appreciate the practical necessity of balancing ideal outcomes with achievable deliverables and reinforced the value of the principle that, at times, “done is better than perfect.”

# Conclusion

This project has demonstrated the feasibility and value of a language-agnostic tool for assessing code quality through design pattern adherence and best practice analysis. By leveraging large language models and a modular backend architecture, the artefact offers structured, adaptable feedback to support software maintainability and clarity. While technical and methodological limitations remain, particularly concerning model reliability and subjective quality metrics, the system provides a practical foundation for future enhancements. Ultimately, the project contributes to ongoing discussions around the responsible integration of AI in software development and highlights the enduring importance of human-centred tooling in an increasingly automated landscape.

# Bibliography

Adenowo, A.A.A. & Adenowo, B.A. (n.d.). *Software Engineering Methodologies: A Review of the Waterfall Model and ObjectOriented Approach*.

Ali, S., Iqbal, M.Z. & Farooq, M.S. (2022). DPVIA: Design Pattern Violations Identification and Assessment Using NLP Techniques. In: *Proceedings of the 2022 International Conference on Software Engineering and Knowledge Engineering (SEKE)*. [Online]. 2022, SEKE, pp. 157–162. Available from: https://doi.org/10.18293/SEKE2022-117.

Alsaqqa, S., Sawalha, S. & Abdel-Nabi, H. (2020). Agile Software Development: Methodologies and Trends. *International Journal of Interactive Mobile Technologies (iJIM)*. 14 (11). p.pp. 246–270.

Amoah, G.Y. (2023). *A Guide to Clean Code: The Power of Good Names*. [Online]. 20 October 2023. DEV Community. Available from: https://dev.to/gervaisamoah/a-guide-to-clean-code-the-power-of-good-names-3f6i. [Accessed: 12 January 2025].

Anthropic (2024). *Claude 2.1 Overview*. [Online]. 2024. Available from: https://www.anthropic.com/index/claude-2-1.

Ayewah, N., Pugh, W., Morgenthaler, J.D., Penix, J. & Zhou, Y. (2008). Using Static Analysis to Find Bugs. In: *IEEE Software*. 2008, pp. 22–29.

Beck, K. (2003). *Test-Driven Development by Example*. Boston: Addison-Wesley.

Beller, M., Gousios, G., Zaidman, A. & Van Deursen, A. (2016). Travis CI: Visualizing Build Failures in Open-Source Software. In: *Proceedings of the 38th International Conference on Software Engineering Companion*. 2016, pp. 103–112.

BetterStack (2025). *Django vs FastAPI: Choosing the Right Python Web Framework*. [Online]. Available from: https://betterstack.com/community/guides/scaling-python/django-vs-fastapi/.

Bosu, A. & Carver, J.C. (2013). Impact of Peer Code Review on Peer Impression Formation: A Survey. In: *Proceedings of the 2013 International Conference on Software Engineering (ICSE)*. 2013, pp. 93–102.

Bryman, A. (2016). *Social Research Methods*. 5th Ed. Oxford: Oxford University Press.

Cabral, B. & Marques, P. (2005). Exception Handling: A Field Study in Java and .NET. *Journal of Object Technology*. 4 (5). p.pp. 149–164.

Chow, Y., Zhao, L. & Zhang, X. (2014). Silent Error Vulnerabilities in Software Systems: Detection, Recovery, and Prevention. *Journal of Systems and Software*. 95. p.pp. 102–116.

Christakis, N. & Spinellis, D. (2020). A Survey on Error Logging Practices in Modern Software Development. *Empirical Software Engineering*. 25 (6). p.pp. 4779–4813.

Codacy (2024). *Codacy: Automated Code Reviews and Analysis*. [Online]. Available from: https://www.codacy.com/.

Creswell, J.W. & Creswell, J.D. (2018). *Research Design: Qualitative, Quantitative, and Mixed Methods Approaches*. 5th Ed. Thousand Oaks: SAGE Publications.

Creswell, J.W. & Plano Clark, V.L. (2011). *Designing and Conducting Mixed Methods Research*. 2nd Ed. Thousand Oaks, CA: SAGE Publications.

DataCamp (2025). *What is Python? Everything You Need to Know to Get Started*. [Online]. Available from: https://www.datacamp.com/blog/all-about-python-the-most-versatile-programming-language.

DeepMind, G. (2024). *Gemini 1.5 Technical Capabilities*. [Online]. 2024. Available from: https://deepmind.google/technologies/gemini/.

Dittrich, Y., Bolmsten, J. & Seidelin, C. (2024). Action Research with Industrial Software Engineering—An Educational Perspective. *arXiv preprint*.

ESLint (2023). *ESLint - Pluggable JavaScript Linter*. [Online]. Available from: https://eslint.org/.

Express js (2025). *Express - Node.js web application framework*.

FastAPI (2025a). *FastAPI*. [Online]. Available from: https://fastapi.tiangolo.com/.

FastAPI (2025b). *Features - FastAPI*. [Online]. Available from: https://fastapi.tiangolo.com/features/.

Feathers, M.C. (2004). *Working Effectively with Legacy Code*. Upper Saddle River, NJ: Prentice Hall.

Flare, C. (n.d.). *Advantages Of Using Python For Data Science*. [Online]. Available from: https://medium.com/@consoleflare/advantages-of-using-python-for-data-science-f751d75a0571.

Fowler, M. & Scott, K. (2000). *UML Distilled: A Brief Guide to the Standard Object Modeling Language*. 2nd Ed. Boston: Addison-Wesley.

Fu, Q., Lou, J.-G., Wang, Y. & Li, J. (2014). Execution Anomaly Detection in Distributed Systems through Unstructured Log Analysis. In: *2014 IEEE International Conference on Data Mining*. 2014, pp. 171–180.

Gamma, E., Helm, R., Johnson, R. & Vlissides, J. (1994). *Design Patterns: Elements of Reusable Object-Oriented Software*. Boston, MA: Addison-Wesley.

GeeksforGeeks (2025a). *SOLID Principles in Programming: Understand With Real Life Examples*. [Online]. 2025. GeeksforGeeks. Available from: https://www.geeksforgeeks.org/solid-principle-in-programming-understand-with-real-life-examples/. [Accessed: 12 January 2025].

GeeksforGeeks (2025b). *Unit Testing – Software Testing*. [Online]. Available from: https://www.geeksforgeeks.org/unit-testing-software-testing/.

George, T. (2024). *What Is Action Research? | Definition & Examples*. [Online]. Available from: https://www.scribbr.com/methodology/action-research/.

GitHub (2024). *GitHub Copilot – Your AI pair programmer*. [Online]. Available from: https://github.com/features/copilot.

Google (2025). *Using files | Gemini API*. [Online]. 2025. Google AI for Developers. Available from: https://ai.google.dev/api/files. [Accessed: 5 February 2025].

Graphite (2024). *Tools and techniques for measuring code quality*. [Online]. Available from: https://graphite.dev/guides/measuring-code-quality-tools-techniques.

Grottke, M. & Trivedi, K.S. (2008). Error Generation and Propagation in Dependable Systems: A Survey. *Reliability Engineering & System Safety*. 94 (1). p.pp. 2–20.

Harm, P. (2018a). SOLID violations in the wild: The Interface Segregation Principle. *Devonblog*. [Online]. Available from: https://devonblog.com/software-development/solid-violations-in-the-wild-the-interface-segregation-principle/. [Accessed: 12 January 2025].

Harm, P. (2018b). SOLID violations in the wild: The Open/Closed Principle. *Devonblog*. [Online]. Available from: https://devonblog.com/software-development/solid-violations-in-the-wild-the-open-closed-principle/. [Accessed: 12 January 2025].

Harm, P. (2018c). SOLID violations in the wild: The Single Responsibility Principle. *Devonblog*. [Online]. Available from: https://devonblog.com/software-development/solid-violations-in-the-wild-the-single-responsibility-principle/. [Accessed: 12 January 2025].

Holmes, S. (2016). *Getting MEAN with Mongo, Express, Angular, and Node*. Manning Publications.

Hombergs, T. (2020). *The Liskov Substitution Principle Explained*. [Online]. 2020. Available from: https://reflectoring.io/lsp-explained/. [Accessed: 12 January 2025].

Institute for Work & Health (IWH) (2015). *Cross-sectional vs. longitudinal studies*. [Online]. Available from: https://www.iwh.on.ca/what-researchers-mean-by/cross-sectional-vs-longitudinal-studies.

JetBrains (2024). *Testing with unittest in PyCharm*. [Online]. Available from: https://www.jetbrains.com/help/pycharm/testing-your-first-python-application.html.

Jin, G., Song, L., Shi, X., Scherpelz, J. & Lu, S. (2012). Understanding and Detecting Real-World Silent Failures in Cloud Systems. In: *USENIX Conference on Operating Systems Design and Implementation (OSDI)*. 2012, pp. 265–278.

Kazman, R., Ozkaya, I. & Yilmaz, C. (2021). Managing Technical Debt in Software-Reliant Systems. *Communications of the ACM*. 64 (7). p.pp. 44–51.

Kazman, R., Ozkaya, I. & Yilmaz, C. (2020). *Technical Debt in Practice: How to Find It and Fix It*. Cambridge, MA: MIT Press.

King, B. (2015). *C# Best Practices - Dangers of Violating SOLID Principles in C#*. [Online]. 1 July 2015. Available from: https://learn.microsoft.com/en-us/archive/msdn-magazine/2014/may/csharp-best-practices-dangers-of-violating-solid-principles-in-csharp. [Accessed: 12 January 2025].

Krekel, F., H.,. Pfannschmidt, R.&. Bruynooghe (2024). *pytest Documentation*. [Online]. Available from: https://docs.pytest.org/en/latest/.

Kuhlman, D. (2020). *A Comparison of Python Testing Frameworks*. [Online]. Available from: https://kuhlman.io/testing-in-python-comparing-unittest-pytest-and-nose.html.

Laravel (2024). *Laravel Pint - PHP code formatter for Laravel projects*. [Online]. Available from: https://laravel.com/docs/pint.

Li, H., Tan, L. & Wang, H. (2019). Learning to Detect Failures: A Study of Static Assertions in Large Systems. *IEEE Transactions on Software Engineering*. 45 (12). p.pp. 1214–1232.

Liu, Y., Le-Cong, T., Widyasari, R., Tantithamthavorn, C., Li, L., Le, X.-B.D. & Lo, D. (2024). Refining ChatGPT-Generated Code: Characterizing and Mitigating Code Quality Issues. *ACM Transactions on Software Engineering and Methodology*. 33 (5). p.p. 116:1-116:26.

López, I., García, M. & Cano, J. (2018). Testing Node.js Applications: Patterns and Tools. *Journal of Software Engineering and Applications*. 11 (3). p.pp. 103–112.

Martin, R.C. (2003). *Agile Software Development: Principles, Patterns, and Practices*. Prentice Hall.

Melnikovas, A. (2018). Adapting Research Onion Model for Futures Studies. *Journal of Futures Studies*. 22 (3). p.pp. 29–44.

Meta (2023). *Jest - Delightful JavaScript Testing*.

Metridev (2024). Code Duplication: Best Practices for Maintainable Programming. *Metridev*. [Online]. Available from: https://www.metridev.com/metrics/code-duplication-best-practices-for-maintainable-programming/. [Accessed: 12 January 2025].

Microsoft (2025). *Pull Request Reviewer*. [Online]. 2025. GenAIScript. Available from: https://microsoft.github.io/genaiscript/guides/pull-request-reviewer/. [Accessed: 5 February 2025].

Multer (2025). *Multer - Node.js middleware for handling multipart/form-data*.

Naumov, A. (2020). *Separation of Concerns in Software Design*. [Online]. 2020. Alexey Naumov. Available from: https://nalexn.github.io/separation-of-concerns/. [Accessed: 12 January 2025].

Newman, L.C. (2024). *Deductive Research*. [Online]. 2024. The Decision Lab. Available from: https://thedecisionlab.com/reference-guide/statistics/deductive-research. [Accessed: 28 April 2025].

Nose2 Developers (2024). *nose2 - Next generation of nicer testing for Python*. [Online]. Available from: https://docs.nose2.io/en/latest/.

Office of Government Commerce (OGC) (2009). *Managing Successful Projects with PRINCE2*. 5th Ed. London: The Stationery Office.

OpenAI (2024). *GPT-4 Technical Report*. [Online]. 2024. Available from: https://openai.com/research/gpt-4.

Oppenheimer, D., Ganapathi, A. & Patterson, D.A. (2003). Why Do Internet Services Fail, and What Can Be Done About It? In: *USENIX Symposium on Internet Technologies and Systems*. 2003, pp. 1–15.

Phair, D. & Warre, K. (2021). *Saunders’ Research Onion Explained (+ Examples)*. [Online]. 26 January 2021. Grad Coach. Available from: https://gradcoach.com/saunders-research-onion/. [Accessed: 28 April 2025].

Projects, P. (n.d.). *Welcome to Flask — Flask Documentation (3.1.x)*. [Online]. Available from: https://flask.palletsprojects.com/.

Python Software Foundation (2024). *unittest — Unit testing framework — Python 3.12.2 documentation*. [Online]. Available from: https://docs.python.org/3/library/unittest.html.

Ray, B., Posnett, D., Filkov, V. & Devanbu, P. (2016). A Large-Scale Study of Programming Languages and Code Quality in GitHub. *Communications of the ACM*. 60 (10). p.pp. 91–100.

Refactoring Guru (2014). *Code Smells*. [Online]. 2014. Available from: https://refactoring.guru/refactoring/smells. [Accessed: 12 January 2025].

Refactoring Guru (2025). *Factory Method*. [Online]. 2025. Available from: https://refactoring.guru/design-patterns/factory-method. [Accessed: 12 January 2025].

Rigby, P.C. & Storey, M.A. (2011). Understanding Broadcast Based Peer Review on Open Source Software Projects. In: *Proceedings of the 33rd International Conference on Software Engineering (ICSE)*. 2011, pp. 541–550.

Royce, W.W. (1970). *Managing the Development of Large Software Systems*. [Online]. Proceedings of IEEE WESCON. Available from: https://www.cs.umd.edu/class/spring2003/cmsc838p/Process/waterfall.pdf.

Saunders, M.N.K., Lewis, P. & Thornhill, A. (2019). *Research Methods for Business Students*. 8th Ed. Harlow: Pearson Education Limited.

Schwaber, K. & Sutherland, J. (2020). *The Scrum Guide*. [Online]. Available from: https://scrumguides.org/scrum-guide.html.

Seref, B. & Tanriover, O. (2016). Software Code Maintainability: A Literature Review. *International Journal of Software Engineering & Applications*. 7 (3). p.pp. 69–87.

Shah, H., Walia, G. & Bishop, J. (2022). Identifying and Classifying Software Error Handling Patterns and Anti-patterns. *Empirical Software Engineering*. 27 (1). p.pp. 1–34.

Shehzad, Z.A. (2023). Building Better Software Systems: How Interfaces and Abstractions Improve Code Quality. *Medium*. [Online]. Available from: https://medium.com/@ZuraizAhmedShehzad/building-better-software-systems-how-interfaces-and-abstractions-improve-code-quality-eb0c85327c24. [Accessed: 12 January 2025].

Sommerville, I. (2016). *Software Engineering*. 10th Ed. Harlow: Pearson Education Limited.

SonarSource (2024). *SonarQube Documentation*. [Online]. Available from: https://docs.sonarqube.org/latest/.

Sonarsource (2025). *What is a Code Smell? Definition Guide & Examples*. [Online]. 2025. Available from: https://www.sonarsource.com/learn/code-smells/. [Accessed: 12 January 2025].

Sourcery AI (2024). *Sourcery - Automated Python Refactoring*. [Online]. Available from: https://sourcery.ai/.

Stack Exchange users (2011). *How should I manage a team with different skill levels?* [Online]. Available from: https://softwareengineering.stackexchange.com/questions/47305/how-should-i-manage-a-team-with-different-skill-levels.

Stony Brook Medicine (2010). *Cross-sectional Versus Longitudinal Study Designs*. [Online]. Available from: https://renaissance.stonybrookmedicine.edu/sites/default/files/Cross-sectional-vs-longitudinal-study-designs.pdf.

Storey, M.A., Zagalsky, A., Filho, F.F., Singer, L. & German, D.M. (2017). How Social and Communication Channels Shape and Challenge a Participatory Culture in Software Development. *IEEE Transactions on Software Engineering*. 43 (2). p.pp. 185–204.

Streefkerk, R. (2019). *Inductive vs. Deductive Research Approach | Steps & Examples*. [Online]. 18 April 2019. Scribbr. Available from: https://www.scribbr.com/methodology/inductive-deductive-reasoning/. [Accessed: 28 April 2025].

TechBullion (2024). *Unpacking the Layers: Understanding the Research Onion*. [Online]. 9 February 2024. TechBullion. Available from: https://techbullion.com/unpacking-the-layers-understanding-the-research-onion/. [Accessed: 28 April 2025].

Thesismind (2019). Analysis of Saunders Research Onion. *Thesismind*. [Online]. Available from: https://thesismind.com/analysis-of-saunders-research-onion/. [Accessed: 28 April 2025].

Tilkov, S. & Vinoski, S. (2010). Node.js: Using JavaScript to Build High-Performance Network Programs. *IEEE Internet Computing*. 14 (6). p.pp. 80–83.

Verywell Mind (2023). *What Is a Longitudinal Study?* [Online]. Available from: https://www.verywellmind.com/what-is-longitudinal-research-2795335.

Vidhya, A. (2021). *Understanding Flask Framework: Installation & features*. [Online]. Available from: https://www.analyticsvidhya.com/blog/2021/10/flask-python/.

Yetiştiren, B., Özsoy, I., Ayerdem, M. & Tüzün, E. (2023). Evaluating the Code Quality of AI-Assisted Code Generation Tools: An Empirical Study on GitHub Copilot, Amazon CodeWhisperer, and ChatGPT. *arXiv preprint arXiv:2304.10778*. [Online]. Available from: http://arxiv.org/abs/2304.10778.

Yin, R.K. (2018). *Case Study Research and Applications: Design and Methods*. 6th Ed. Thousand Oaks, CA: SAGE Publications.

Zampetti, F., Di Penta, M., Oliveto, R. & Canfora, G. (2017). An Empirical Characterization of Bad Practices in Continuous Integration. *Empirical Software Engineering*. 22. p.pp. 256–289.

Zhang, X., Zhang, D., Li, Z., Wu, J. & Zhang, Y. (2022). A Resilience-Oriented Survey of Software Fault-Tolerance Techniques. *Journal of Systems and Software*. 189. p.p. 111313.

Zheng, A.X., Neylon, E., Bao, M. & Liu, M.Y. (2011). Automatic Failure Diagnosis using Graph-Based Patterns. *Communications of the ACM*. 54 (7). p.pp. 54–62.

# Appendices

## Appendix A – Project Plan

The management of this project will utilise "Shortcut," a project management tool chosen for its flexibility and alignment with Agile principles. Each major objective within the project will be divided into clear, actionable tasks to ensure that every component is approached systematically. This breakdown of tasks will enable a precise focus on individual project elements, helping to track progress and ensure that no critical steps are overlooked.

Tasks will be organised into specific iterations, each with a defined duration. By setting projected timeframes for each iteration, the project can maintain a steady pace and meet key milestones efficiently. This structured organisation will not only support the workflow but also facilitate regular reviews, allowing adjustments as necessary. The use of iterations will also support ongoing assessments of project progress and quality, helping to align outcomes with both the project goals and Agile methodology standards.

Within Shortcut, features such as milestone markers and progress tracking provide valuable insights into the project's overall status. This visibility allows for early identification of any potential roadblocks, ensuring that issues can be addressed proactively. Additionally, Shortcut’s adaptable framework complements the iterative nature of this project, which requires continuous feedback and adaptation. By incorporating these Agile principles into the planning and execution phases, the project can remain responsive to evolving requirements and focus on delivering a well-rounded final product that meets its intended objectives.

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

## Appendix B – Health and Safety Risk Assessment

| Staffordshire University Logo**GENERAL RISK ASSESSMENT FORM** | | | **Severity** multiplied by **Likelihood** equals **Risk Rate**.  NB: Calculated after taking in to account existing precautions | | | | |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Department: DTIB | | Severity | Insignifi-cant (1) | Minor (2) | Moder-ate (3) | Serious (4) | Fatal / Critical (5) |
| Likelihood |
| Task/Activity/Area: Final Year Project | | Almost Certain (5) | **5** | **10** | **15** | **20** | **25** |
| Likely (4) | **4** | **8** | **12** | **16** | **20** |
| Student ID: 22042944 | Signature: | Possible (3) | **3** | **6** | **9** | **12** | **15** |
| Unlikely (2) | **2** | **4** | **6** | **8** | **10** |
| Date of Assessment: 01/11/2024 | Review Date: 01/11/2024 | Rare (1) | **1** | **2** | **3** | **4** | **5** |

| **ID** | **Activity/Process/Machines** | **Hazard** | **Persons in Danger** | **Severity 1-5** | **Likelihood 1-5** | **Risk Rate** | **Measures/Comments** | **Result** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | Failure to disconnect and treat appropriate devices carefully | Electrocution | Myself, User | 4 | 1 | 4 | Very unlikely to occur. Basic health and safety precautions and fail-safes should already be in place. Avoid damaging any cables or contact with live wires. | T |
| 2 | Looking and using at electronic screens for extended periods of time | Eye strain / RSI | Myself, User | 2 | 1 | 2 | Taking breaks every hour to prevent any potential issues and allow both eyes and wrists to rest. | T |
| 3 | Data gets lost or corrupted | Loss of data | Myself | 5 | 1 | 5 | Back up data at regular intervals | T |
| 4 | Sickness preventing work | Contracting infection or developing a debilitating condition | Myself | 4 | 1 | 4 | Avoid obvious infection vectors | A |

Key to result **T** = Trivial Risk **A** = Adequately Controlled **N** = Not Adequately Controlled **U** = Unable to decide (further information required).

## Appendix C – Ethical Statement

My ethical statement document will be uploaded alongside this document, but images of the form are included below.

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

All key documents e.g., consent form, information sheet, questionnaire/interview schedules will be checked by supervisor before any study commences.

Ethical Statement: I am committed to adhering to the BCS Code of Conduct and all relevant GDPR regulations throughout this project. All participants will be informed about the nature of the information collected and their rights, including the option to request the removal of their data at any time. I will ensure participants are fully briefed and understand the project's purpose and intent before their involvement, and they will be made aware that they can withdraw from the project at any stage.