CONTENT 2025

The Seventeenth International Conference on Creative Content Technologies

ISBN: 978-1-68558-262-3

April 6 - 10, 2025

Valencia, Spain

CONTENT 2025 Editors

Steve Chan, VTIRL, VT/DE-STEA - Orlando, USA

CONTENT 2025

Forward

The Seventeenth International Conference on Creative Content Technologies (CONTENT 2025), held on April 6 – 10, 2025, continued a series of events targeting advanced concepts, solutions and applications in producing, transmitting and managing various forms of content and their combination. Multi-cast and uni-cast content distribution, content localization, on-demand or following customer profiles are common challenges for content producers and distributors. Special processing challenges occur when dealing with social, graphic content, animation, speech, voice, image, audio, data, or image contents. Advanced producing and managing mechanisms and methodologies are now embedded in current and soon-to-be solutions.

Similar to the previous edition, this event attracted excellent contributions and active participation from all over the world. We were very pleased to receive top quality contributions.

We take here the opportunity to warmly thank all the members of the CONTENT 2025 technical program committee, as well as the numerous reviewers. The creation of a high quality conference program would not have been possible without their involvement. We also kindly thank all the authors that dedicated much of their time and effort to contribute to CONTENT 2025. We truly believe that, thanks to all these efforts, the final conference program consisted of top quality contributions.

Also, this event could not have been a reality without the support of many individuals, organizations and sponsors. We also gratefully thank the members of the CONTENT 2025 organizing committee for their help in handling the logistics and for their work that made this professional meeting a success.

We hope CONTENT 2025 was a successful international forum for the exchange of ideas and results between academia and industry and to promote further progress in the area of creative content technologies. We also hope that Valencia provided a pleasant environment during the conference and everyone saved some time to enjoy this beautiful city.

CONTENT 2025 General Chair

Hans-Werner Sehring, NORDAKADEMIE - University of Applied Sciences, Germany

CONTENT 2025 Steering Committee

Raouf Hamzaoui, De Montfort University - Leicester, UK
Manfred Meyer, Westfälische Hochschule, Germany
Mu-Chun Su, National Central University, Taiwan

CONTENT 2025 Publicity Chair

Francisco Javier Díaz Blasco, Universitat Politècnica de València, Spain
Ali Ahmad, Universitat Politècnica de València, Spain

José Miguel Jiménez, Universitat Politècnica de València, Spain
Sandra Viciano Tudela, Universitat Politècnica de València, Spain

CONTENT 2025

Committee

CONTENT 2025 General Chair

Hans-Werner Sehring, NORDAKADEMIE - University of Applied Sciences, Germany

CONTENT 2025 Steering Committee

Raouf Hamzaoui, De Montfort University - Leicester, UK
Manfred Meyer, Westfälische Hochschule, Germany
Mu-Chun Su, National Central University, Taiwan

CONTENT 2025 Publicity Chair

Francisco Javier Díaz Blasco, Universitat Politècnica de València, Spain
Ali Ahmad, Universitat Politècnica de València, Spain
José Miguel Jiménez, Universitat Politècnica de València, Spain
Sandra Viciano Tudela, Universitat Politècnica de València, Spain

CONTENT 2025 Technical Program Committee

Kambiz Badie, Research Institute for ICT & University of Tehran, Iran
René Berndt, Fraunhofer Austria Research GmbH, Austria
Mario Bisson, Politecnico di Milano, Italy
Christos J. Bouras, University of Patras, Greece
Vincent Charvillat, Toulouse University, France
Xuanbai Chen, Amazon, USA
Juan Manuel Corchado Rodríguez, Universidad de Salamanca, Spain
Rafael del Vado Vírseda, Universidad Complutense de Madrid, Spain
Sotiris Diplaris, Information Technologies Institute - Centre for Research and Technology Hellas, Greece
Jimmy Eadie, Trinity College Dublin, Ireland
Hannes Fassold, Joanneum Research - Digital, Graz, Austria
Joshua A. Fisher, Columbia College Chicago, USA
Valérie Gouet-Brunet, IGN / LaSTIG, France
Raouf Hamzaoui, De Montfort University, UK
Yuntian He, The Ohio State University, USA
Verena Kantere, National Technical University of Athens, Greece / University of Ottawa, Canada
Wen-Hsing Lai, National Kaohsiung University of Science and Technology, Taiwan
Alain Lioret, Université Paris 8, France
Junhua Liu, The Chinese University of Hongkong, Shenzhen, China
Yong Liu, Universiti Brunei Darussalam, Brunei
Nadia Magnenat-Thalmann, University of Geneva, Switzerland & Nanyang Technological University,

Copyright Information

For your reference, this is the text governing the copyright release for material published by IARIA.

The copyright release is a transfer of publication rights, which allows IARIA and its partners to drive the dissemination of the published material. This allows IARIA to give articles increased visibility via distribution, inclusion in libraries, and arrangements for submission to indexes.

I, the undersigned, declare that the article is original, and that I represent the authors of this article in the copyright release matters. If this work has been done as work-for-hire, I have obtained all necessary clearances to execute a copyright release. I hereby irrevocably transfer exclusive copyright for this material to IARIA. I give IARIA permission or reproduce the work in any media format such as, but not limited to, print, digital, or electronic. I give IARIA permission to distribute the materials without restriction to any institutions or individuals. I give IARIA permission to submit the work for inclusion in article repositories as IARIA sees fit.

I, the undersigned, declare that to the best of my knowledge, the article is does not contain libelous or otherwise unlawful contents or invading the right of privacy or infringing on a proprietary right.

Following the copyright release, any circulated version of the article must bear the copyright notice and any header and footer information that IARIA applies to the published article.

IARIA grants royalty-free permission to the authors to disseminate the work, under the above provisions, for any academic, commercial, or industrial use. IARIA grants royalty-free permission to any individuals or institutions to make the article available electronically, online, or in print.

IARIA acknowledges that rights to any algorithm, process, procedure, apparatus, or articles of manufacture remain with the authors and their employers.

I, the undersigned, understand that IARIA will not be liable, in contract, tort (including, without limitation, negligence), pre-contract or other representations (other than fraudulent misrepresentations) or otherwise in connection with the publication of my work.

Exception to the above is made for work-for-hire performed while employed by the government. In that case, copyright to the material remains with the said government. The rightful owners (authors and government entity) grant unlimited and unrestricted permission to IARIA, IARIA's contractors, and IARIA's partners to further distribute the work.

Table of Contents

Integrating Creative Artifacts into Software Engineering Processes

Hans-Werner Sehring
Department of Computer Science
Nordakademie
Elmshorn, Germany
e-mail: sehring@nordakademie.de

***Abstract*—Model-driven software engineering processes are based on formal models that are automatically transformed into each other. Many software development approaches involve creative activities that result in manually generated and informal documents that prevent automatic model transformations. The content of these documents must be accessed in a structured way to enable transformation steps. Manually maintained documents are subject to frequent changes, including modifications of their structure. To enable model-driven processes in the presence of creative activities and their documents, we are currently experimenting with parsing techniques that combine the structure of documents with domain knowledge about their content. First experiments are based on the Minimalistic Metamodeling Language and its ability to integrate semantic descriptions with syntactic representations.**

Keywords-software development; software engineering; computer aided software engineering; top-down programming; document handling.

I. Introduction

Software engineering processes involve the creation and consumption of a series of documents. Such documents link different phases of activity in software creation processes, be they sequential work performed by experts in phase-oriented projects or simultaneous cooperation in cross-functional teams in agile approaches.

A class of software engineering processes that are based on documents that contain formal models are called *Model-Driven Software Engineering* (*MDSE*) or *Model-Driven Software Development* (*MDSD*) processes.

Some software engineering processes include creative activities, such as conceptual modeling or interaction design [1]. Such creative actitivies are supported by documents that have neither a common format [2] nor formal semantics. Instead, they reflect subjective impressions, case-based presentations, alternatives, and similar content directed at a human audience.

Documents that lack formal structure cannot participate in MDSE processes per se. However, they can be annotated by their creators with, for example, with references to relevant content that are sufficiently fine-grained to address well-formed content. Such annotations allow creative documents to participate in MDSE processes.

However, such annotations refer to specific document instances. Documents used in creative activities are, in particular, working documents that are subject to constant change. This includes changes in the structure of the documents. Therefore, any fixed reference to content in such a document will potentially become invalid and metadata may become inconsistent as work progresses.

In this paper, we investigate means of integrating informal documents, in particular ones that are subject to change, into (model-driven) software engineering processes. We are currently experimenting with linguistic means of recognizing the content of documents with changing structures. First experiments with document recognition are based on a modeling language and its special ability to integrate semantic descriptions with syntactic representations.

Preliminary results show that at least some content can be extracted from documents that lack formal representations. In this way, model-driven approaches can potentially be applied to software projects with creative aspects.

The remainder of this paper is organized as follows: In Section II, we revisit model-driven software engineering and discuss the need for incorporating informal documents. Section III presents typical ways of referencing content in single documents, and it addresses means of managing volatile references to content of mutable documents. Section IV briefly introduces a modeling language that is used for initial experiments in this paper. An experimental implementation of these concepts is presented in Section V. The paper concludes in Section VI with a summary and an outlook on future work.

II. Visual Software Engineering Artifacts

The discourse in this paper does not require a comprehensive introduction to model-driven approaches. However, this section introduces some basic terms and highlights the challenges of integrating creative work.

A. Model-Driven Software Engineering

In software development processes, a series of documents is created. The kinds of documents may differ depending on the kind of software being created and on the methodology used for the process. But all documents serve common purposes, such as linking activities by the results represented in them, allowing traceability of activities [3], and others.

MDSE formalizes the flow of documents and thus the connection of development steps. Documents are *models* with a formal semantics. Models are derived by means of *model-to-model transformations* and finally to code in *model-to-text transformations* on a (semi-) automatic basis. This way, development steps can be performed (semi-) automatically and changes to models can be propagated down the model chain.

One of the first prominent examples of MDSE is the Object Management Group's Model-Driven Architecture (MDA). Various other approaches have emerged that differ in the way in

 ISBN: 978-1-68558-262-3

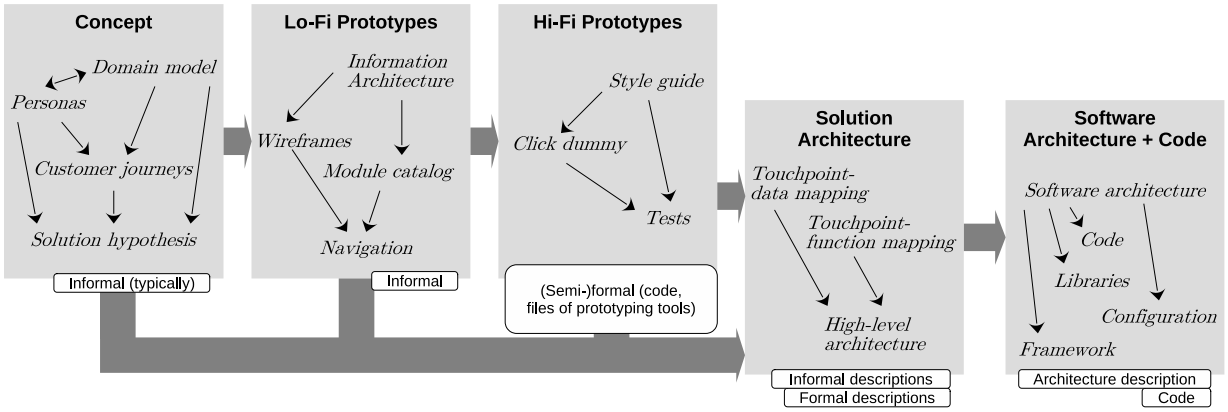


Figure 1. A typical software development process that integrates creative activities [4].

which they implement transformations, for example, by means of metaprogramming [5], code templates [6], or generative artificial intelligence [7].

B. Creative Software Development Activities

Certain kinds of software solutions, for example, one with a focus on the human-machine interface, include creative steps. Examples of creative activities are the definition of interaction patterns, of user experience in general, and user interface design in particular.

In [8], we use the term *Model-Supported Software Creation* (*MSSC*) to distinguish this kind of software development from general MDSE that relies purely on formal representations.

Figure 1 shows typical development steps and artifacts created to model aspects of a software solution. Models, such as the *Domain model*, the *High-level architecture*, and the *Software architecture* can typically be expressed in a suitably formal way as to be derived from each other by model-to-model transformations. However, other documents are typical representatives of informal documents, such as *Personas*, *Customer journeys*, and *Style guides*. There may even be dynamic artifacts, such as a *click dummy* that needs to be experienced by a human observer who interacts with it.

C. Creative Artifacts in Model-Driven Processes

Depending on the type of software, there are different steps in the development process that are of an informal nature. Some software solutions require creative development activities. Typical such activities are those from the disciplines of domain modeling, conceptual modeling, and visual design. Such development steps are typically performed manually and lead to subjective results. As a result, tools that support creative activities often produce informal representations and documents. Therefore, software projects that involve creative activities cannot be fully covered by model-driven processes in most approaches.

In order to include creative actitivies in model-driven processes, the informal documents that are generated have to be interpreted in such a way that their content can be referenced and can be extracted in a defined structure. Through such an interpretation, content may be used in software models or during model transformations.

Interpretations of documents that lack formal structure can be added explicitly. For example, their creators may provide annotations with *content references* and *metadata* to guide access to relevant content. Such annotations, however, refer to specific document instances.

Creative activities typically consist of numerous iterations. As a consequence, documents used in creative activities are subject to constant change. Changes include changes to the structure of the documents. Therefore, any fixed reference to content in such a document will potentially become invalid and metadata may become inconsistent as work progresses. As a consequence, documents are required to be constantly reinterpreted.

III. Referencing Content in Documents

In order to extract content from documents in a form that is suitable for use in a formal development process, parts of that content must be addressable. This requires documents to be structured, or to allow superimposed structures for content references.

Digital documents can be structured to varying degrees. Typically, document formats are categorized as *structured*, *semi-structured*, and *unstructured*.

A. Structured Documents

Structured documents are created according to a well-defined structure and they can be analyzed precisely according to that structure. This can be realized in three different ways. The structure of documents may be used to query for content, such as object paths based on JSON definitions. To be able to address specific parts of a document, structure elements must have stable names (paths) or stable IDs. A different approach is grammars, which can be used both to create documents of a certain form and to parse documents to identify structural elements according to linguistic constructs.

A common structure to which multiple documents conform calls for a schema or document format. Schemas of structured documents differ in the meaning they convey. A format may

reflect visual layout like, for example, in the case of HTML, it may use a generic semantics like, for example, XML formats for formal languages, or it may carry domain knowledge as, for example, application-specific XML formats.

B. Semistructured Documents

Documents that have a recognizable structure, but no common schema to which they conform, are called *semistructured*. Any interpretation rules applied to such documents are fragile in the sense that they may not be applicable to all document instances, or else all possible forms of documents must be considered.

If there is some technical structure that allows referencing parts of a document, then some pragmatics can be applied to interpret combinations of structure elements and content. For example, in a text document, there may be a recognizable structure of single-line terms written entirely in bold font. That may be interpreted as the term being a section heading. If the document is a software architecture description, and if the term is interpreted as a subsection of a section "Software Components", then the term may be interpreted as the name of a software component.

In this way, semistructured documents are required to expose some recognizable structure, and interpreting them requires some known domain semantics and pragmatics to apply some interpretation rules.

C. Unstructured Documents

Unstructured documents, exhibit no structure that would allow referencing parts of a document. Typical examples of such documents are media files in binary format.

To reference parts of an unstructured document, some technical ways of addressing can be used, for example, pixel ranges in an image or timecode sequences in movies. Such references depend on the concrete document or, more precisely, on the actual presentation of it. For example, areas of an image that are defined by pixel coordinates relate to the resolution of that image. Such references are, therefore, volatile. For example, a selection of pixel coordinates is not valid for an equivalent image in different resolution.

There is no precise way to semantically reference content, although the semantics of unstructured documents can be analyzed by various algorithms.

D. Aggregating Documents from Different Sources

When accessing document collections that originate from different sources, the problem of different or varying schemas may aries. A typical approach to cope with such a situation is employing adapter components that allow accessing structured documents according to a common schema or by transforming them into a common schema [9].

E. Extracting Content from Mutable Documents

As mentioned earlier, documents created during creative activities in software engineering processes are subject to change, which means they have to be mutable (volatile, sometimes called *living* documents).

In MDSE processes, the contents of documents are used to create software models from them, or such models are in other ways related to the contents of documents. Changing documents can generally break such relationships.

One solution is to create copies of documents once they are referenced and to keep these copies stable. But this would exclude further work on those documents from the process.

Parsing is a standard approach to identifying meaningful content in a document. For formal languages, a parsing process operates on the syntactic structures of a document and applies a defined semantics to interpret those structures. Documents resulting from creative processes do not follow a fixed semantics. Therefore, classical parsing approaches based on formal languages alone do not work on them. In our current research, we augment document parsing with the application of domain knowledge.

Parsing of semistructured documents requires pragmatics since not all parts of the document have an identifiable structure. An open question is whether pragmatics can be provided by domain knowledge: two equally formatted expressions may be distinguished by some significant content. In general, domain knowledge may be necessary to decide on a parsing strategy.

Parsing is well understood for formal and, to a limited extent, semistructured representations, but it is usually applied once. Updating models based on subsequent parsing results of a modified document requires, according to our current findings, an additional relationship between document structure and domain semantics.

IV. The M³L as a Modeling Language

The *Minimalistic Metamodeling Language*, short *M³L*, is a metamodeling language. As such, it can be employed for models for different kinds of applications. We use it for first experiments in document recognition by capturing domain semantics as well as document formatting.

The M³L allows defining and deriving *concepts*. Definitions are of the general form

A is a **B** { **C** is a **D** } |= **E** { **F** } |- **G H** .

Such a statement matches or creates a concept *A*. All parts of such a statement except the concept name are optional.

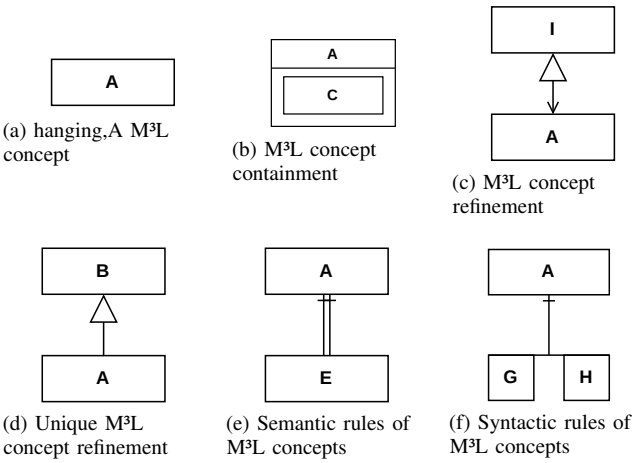
In the course of this paper we use a graphical notation of the M³L as shown in Figure 2 for the different parts of a concept definition. For concept refinement we borrow notation from the *Unified Modeling Language* (*UML*), see Figure 2c for *is a* relationships and Figure 2d for *is the* relationships.

The concept *A* is a *refinement* of the concept *B*. Using the "is the" clause instead defines a concept as the only specialization of its base concept.

The concept *C* is defined in the *context* of *A*; *C* is part of the *content* of *A*. Contexts define (hierarchical) scopes. Concepts, such as *A* are defined in an unnamed top-level context.

There can be multiple statements about a concept visible in a scope. Statements about a concept are cumulated. This allows concepts to be defined differently in different contexts.

For an example of modeling with the the M³L, consider the definition of a conditional statement found in imperative



(a) hanging,A M³L concept
(b) M³L concept containment
(c) M³L concept refinement
(d) Unique M³L concept refinement
(e) Semantic rules of M³L concepts
(f) Syntactic rules of M³L concepts

Figure 2. A graphical notation of M³L concepts.

```
ConditionalStatement is a Statement {
  Condition     is a Boolean
  ThenStatement is a Statement
  ElseStatement is a Statement }
```

Figure 3. Sample base model of procedural programming.

```
IfTrueStmt is a ConditionalStatement {
  True is the Condition
} |= ThenStatement
IfFalseStmt is a ConditionalStatement {
  False is the Condition
} |= ElseStatement
```

Figure 4. Sample semantics of conditional statements.

programming languages in Figure 3. It consists of *Condition* to decide whether to execute *ThenStatement* or *ElseStatement*.

Semantic rules can be defined on concepts, denoted by "|=". A semantic rule references another concept, that is returned when a concept with a semantic rule is referenced. Like for any other reference, a non-existent concept is created on demand.

Context, specializations, and semantic rules are employed for *concept evaluation*. A concept evaluates to the result of its syntactic rule, if defined, or itself, otherwise. Syntactic rules are inherited from explicit base concepts (given by *is a*/*is the*) and implicit base concepts (concepts with matching content).

By means of concept evaluation, semantics can be assigned to concepts. The code in Figure 4 uses syntactic rules to assign semantics to the conditional statement from the example above. A concrete statement is matched against the two subconcepts *IfTrueStmt* and *IfFalseStmt*. If one of them is recognized as a derived base concept of the given statement, the semantic rule of the matching concept is inherited. This way, the "then" statement or the "else" statement is executed (evaluated next).

Concepts can be marshaled/unmarshaled as text by *syntactic rules*, denoted by "|-". A syntactic rule names a sequence of concepts whose representations are concatenated. A concept without a syntactic rule is represented by its name. Syntactic

```
Java is a ProgrammingLanguage {
 ConditionalStatement
  |- if ( Condition ) ThenStatement
     else ElseStatement . }
Python is a ProgrammingLanguage {
 ConditionalStatement
  |- if Condition :
     "\n  " ThenStatement
     else:
     "\n  " ElseStatement . }
```

Figure 5. Sample syntax of the conditional statement.

rules are used to represent a concept as a string as well as to create a concept from a string.

Figure 5 shows syntactic rules that map the conditional statement from the example to different programming languages.

V. First Experiments Using the M³L

Describing static documents with metadata provided as concepts that make reference to relevant parts of the content has been researched in the past. Some initial experiments with simple documents have been conducted to investigate means of linguistic document interpretation.

A. Static Document References

As a first example of document descriptions using the M³L, Figure 6 illustrates static references to (fragments of) documents. It uses an example from art history. A picture of a painting shown on the bottom of Figure 6 is described using (M³L) concepts.

The concept hierarchy starting with the concept *DocumentReference* defines references to (fragments of) documents. A *DocumentId* defines some address of a document (file name, URL, or similar), and *FragmentSelector* defines a part of a document that holds interesting content. For the example, we see a sketch of a refinement hierarchy which specifies concepts for references to two-dimensional images, for those depicting paintings, and paintings that specifically show a ruler.

A second concept hierarchy starting with *DocumentDescription* contains concepts that describe the subject of a document. The two hierarchies meet at the *PaintingDescription*. An application-specific concept RulerPaintingDescription refines it for the area of interest, and *NapoleonCrossesTheAlps* finally provides an "instance" of a ruler painting.

B. Interpretation of Semistructured Documents

As a foundation of the interpretation of some kind of documents, some general concepts are defined first. Figure 7 shows an example of documents that represent customer journeys and that are exported from a (hypothetical) whiteboard software. A concept *Board* allows to reference a whiteboard, a concept *Page* some page (assuming the whiteboard software allows to subdivide whiteboards). On a whiteboard, there is no recognizable structure below the page level. Starting with the concept *CustomerJourney*, we look for semantic structures on a

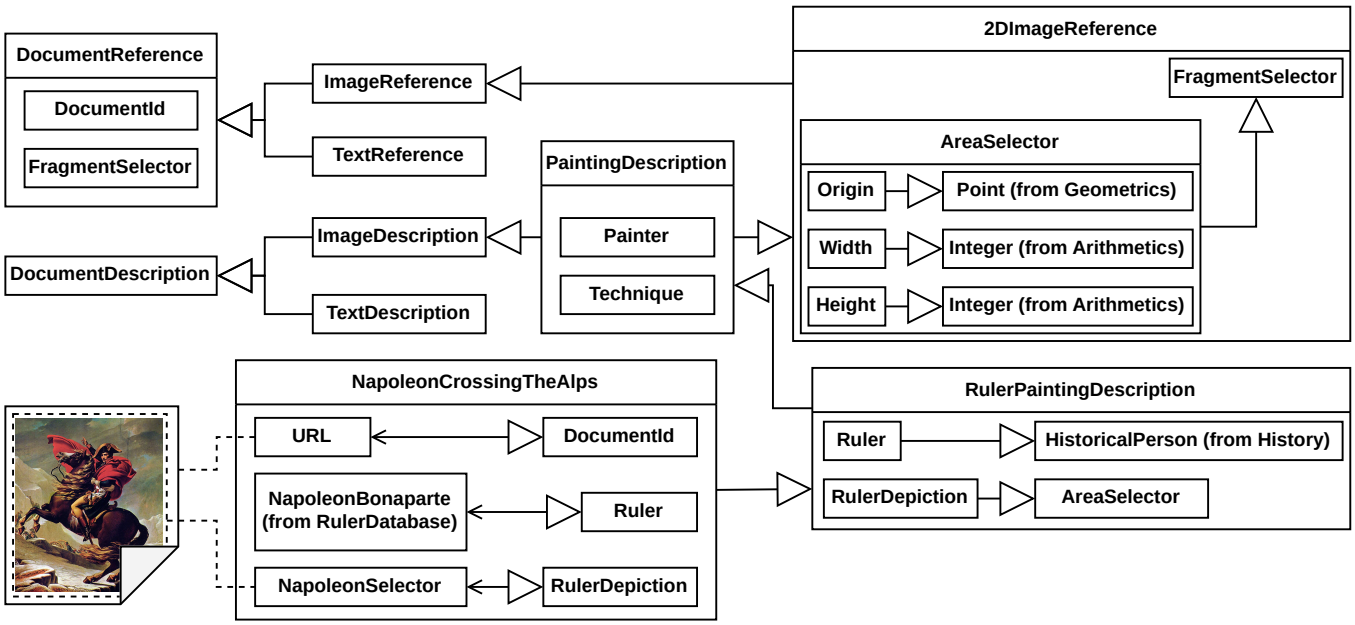


Figure 6. Static references to documents and document fragments.

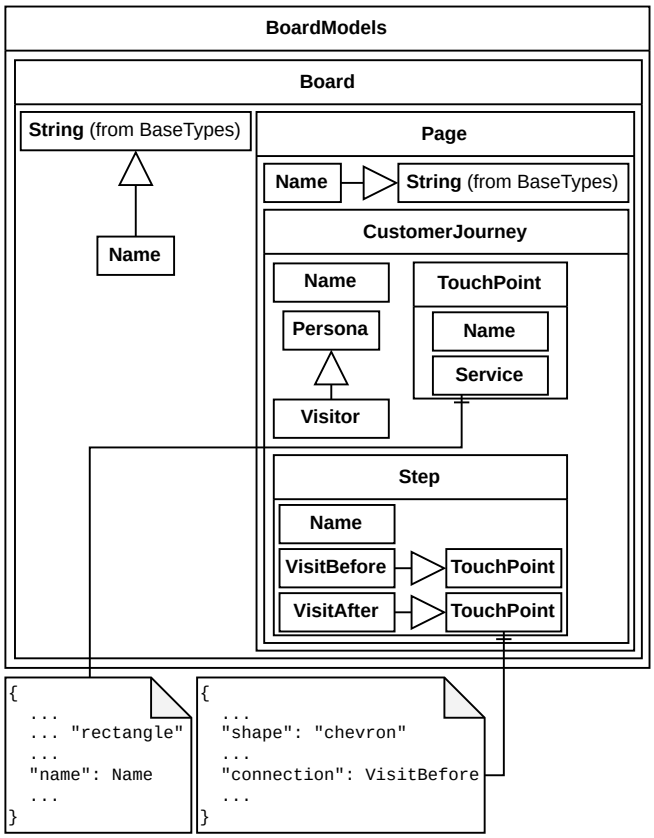


Figure 7. Example of a pattern for mutable documents.

whiteboard page. A customer journey is some named (graphical) object that consists of elements that represent *Touchpoint*s and ones that represent *Step*s. A touchpoint is characterized by a *Name* and a *Service*. Syntactic rules define how such concepts are represented on a whiteboard page. Figure 7 sketches some rules that generate/recognize JSON code as it might come out of a whiteboard software that is provided as a Cloud service.

Once a customer journey has been developed on a whiteboard of that form, the syntactic rules can be used to recognize the structure and to extract the content of it. Figure 8a shows a sample customer journey. Also in the example of that figure, a (M³L) concept for the board has been created as a subconcept of *Board* from Figure 7 with a reference to the board document.

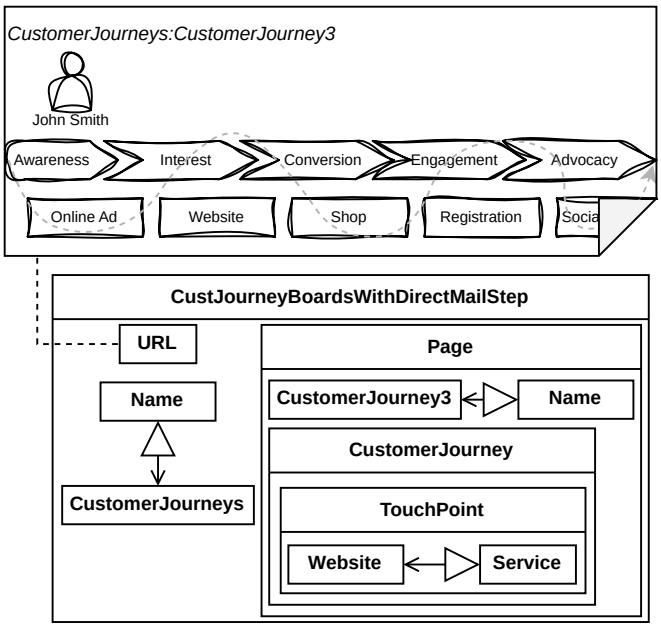
When the board is interpreted according to the syntactic rules for *Board*s, the result is the concept structure from Figure 8b. The concepts that have been created from the board reflect some of the design decisions contained in the customer journey representation, such as the participating persona and the relationships to the touchpoints it visits and the sequence of touchpoints along the customer journey.

The extracted information can be used in subsequent activities of the software development process. Using the M³L, the resulting concepts can be related to concepts that represent models created in such subsequent activities.

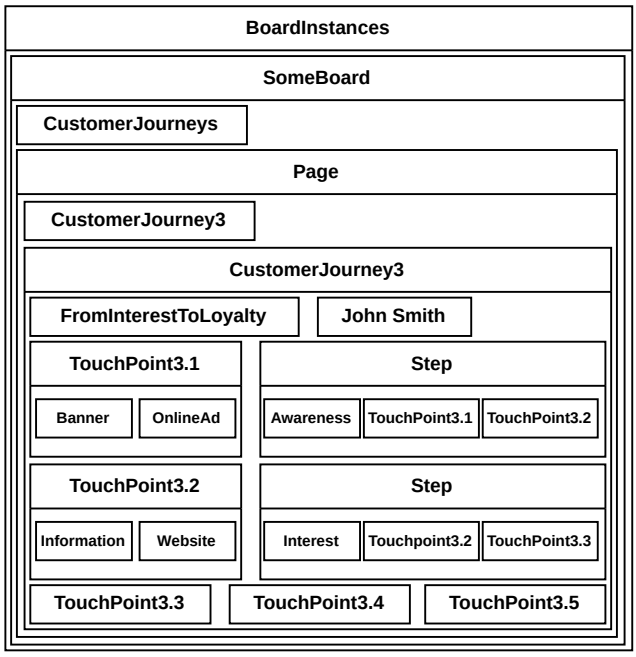
C. Reinterpretation of Mutable Documents

Mutable documents are handled by repeatedly applying the parsing process. When reinterpreting a document after a change, fresh concept definitions are made in the M³L. Due to M³L's way of matching definitions against existing concepts before creating new ones, previous interpretations are found and used in the parsing process. Depending on the concept model, existing concept references that were established by model-to-model transformations are preserved. In this way, documents can be modified even if they have already been interpreted and related to other models during an MDSE process.

 ISBN: 978-1-68558-262-3



(a) Example of a query to mutable documents.



(b) Example result of mutable document recognition.

Figure 8. Parsing of documents and document fragments.

Recognition of existing concepts requires some stable information. These may, for example, be unique names as well as a certain location in the document structure where it is placed. In the example of the digital whiteboards above, names might be given in a specially positioned text field. As a consequence, the documents are not completely mutable, at least not in terms of content.

An agreement on some recognizable information constitutes a restriction to the idea of mutable documents. Finding ways of leveraging this situation is subject to future work.

VI. Conclusion and Future Work

In this paper, we investigate an approach to integrate semi-structured documents supporting creative activities into MDSE processes. Using the M³L, documents can be parsed based on their syntactic structure in conjunction with the semantics of the concepts represented in such documents. A first simple experiment shows that content can be extracted from a document in a suitably formal form if the document follows some conventions. The concepts recognized in a document can serve as model elements that link the documents to the chain of model-to-model transformations of MDSE processes.

Future work will need to test this approach with a range of existing file formats and service APIs to further investigate the limits of document interpretation and possibly identify additional requirements for parsing technology. There are limits to the extent to which documents can be modified without losing existing links to software models. These limits are not well researched. We need to find the limits, ways to extend them, and notations to describe parts of documents that must not be altered. Another future research direction concerns a form of roundtrip engineering in which documents are not only interpreted, but also generated from models that need to be presented in a form suitable for non-technical stakeholders.

Acknowledgement

The author thanks the Nordakademie for granting the opportunity to publish this work.

References

[1] G. Liebel *et al.*, “Human factors in model-driven engineering: Future research goals and initiatives for mde”, *Software and Systems Modeling*, vol. 23, no. 4, pp. 801–819, 2024.

[2] E. Herac, L. Marchezan, W. Assunção, R. Haas, and A. Egyed, “A flexible operation-based infrastructure for collaborative model-driven engineering”, in *Modellierung 2024*, ser. Lecture Notes in Informatics (LNI), Gesellschaft für Informatik e.V., 2024.

[3] I. Galvao and A. Goknil, “Survey of traceability approaches in model-driven engineering”, in *Proceedings of the 11th IEEE International Enterprise Distributed Object Computing Conference*, 2007, pp. 313–313.

[4] H.-W. Sehring, “Visual artifacts in software engineering processes”, in *Proceedings of the Sixteenth International Conference on Creative Content Technologies*, ThinkMind, 2024, pp. 1–6.

[5] S. Trujillo, M. Azanza, and O. Diaz, “Generative metaprogramming”, in *Proceedings of the 6th international conference on Generative programming and component engineering GPCE ’07*, Association for Computing Machinery, 2007, pp. 105–114.

[6] J. Arnoldus, M. Van den Brand, A. Serebrenik, and J. J. Brunekreef, *Code generation with templates*. Springer Science & Business Media, 2012, vol. 1.

[7] K. Lano and Q. Xue, “Code generation by example using symbolic machine learning”, *SN Computer Science*, vol. 4, Jan. 2023.

[8] H.-W. Sehring, “Model-supported software creation: Towards holistic model-driven software engineering”, in *Proceedings of the 2023 IARIA Annual Congress on Frontiers in Science, Technology, Services, and Applications*, ThinkMind, 2023, pp. 113–118.

[9] I. Amous, A. Jedidi, and F. Sèdes, “A contribution to multimedia document modeling and querying”, *Multimedia Tools and Applications*, vol. 25, pp. 391–404, 3 Oct. 2005.

The Generation of Piano Music in the Style of Johannes Brahms Using Neural Network Architectures

James Doherty
Technological University Dublin
Central Quad, Grangegorman Lower, Dublin 7, D07ADY7
Dublin, Ireland
e-mail: d14126143@mytudublin.ie

Brendan Tierney
Technological University Dublin
Central Quad, Grangegorman Lower, Dublin 7, D07ADY7
Dublin, Ireland
e-mail: brendan.tierney@tudublin.ie

Abstract - **Neural network architectures currently are only able to employ music generation tasks to similar levels of human composers when the music is at a basic compositional standard as they struggle with the complex motifs and harmonic structures of Western Classical Music. This study aims to determine if various data preprocessing and augmentation techniques can train a neural network model to generate pieces of piano music to a similar level of musicality and emotion as Romantic Period composer Johannes Brahms. Quantitative experimentation involving Music Information Retrieval was conducted, as well as a quantitative survey with respondents consisting of only professional musicians, composers, and conductors. Analysis of the results demonstrated that Transformer models using various attention mechanisms generated statically similar results to the original piano works of Brahms and that survey participants struggled to distinguish between the pieces generated by Brahms and the models. The results indicate that various data preprocessing and augmentation methods do have an impact on model accuracy resulting in the ability to generate longer sequences of music containing the composite motifs and harmonic structures of romantic period piano music.**

Keywords-artificial Intelligence; music generation; neural network architecture; Brahms.

I. INTRODUCTION

The intention of this project is to generate piano music in the style of classical music composer Johannes Brahms by training a Recurrent Neural Network (RNN), a Long Short-Term Memory (LSTM) based RNN, Transformer models with different attention mechanisms, and a Perceiver AR model. These models will be trained with a pre-processed and augmented dataset containing MIDI files of Brahms' piano works. To deem the success of the project, the best musical pieces generated from the neural network models must show statistical similarities in various musical variables using Music Information Retrieval (MIR). Along with this, the pieces must also be mistaken by professional musicians, composers, and conductors as one of Brahms' own works through a quantitative survey. Although there have been many examples of AI models generating music in the style of particular composers, no models have been created to generate the work of Brahms. The lack of a detailed computational analysis of Brahms shows a gap in the study of romantic period composers, which Brahms was a key figure of [1]. According to studies taken, computer-generated music has traditionally only sounded human-like when short excerpts were created and struggled with the complex motifs and harmonic cadences of romantic period piano music. This could be down to them being poor at handling higher-level musical structures due to the models only learning how to play the next note according to the previous [2]. In their paper, Child et al. developed a sparse transformer and stated that it was able to extract complex patterns from sequences up to 30 times longer than possible previously [3]. Likewise, Hawthrone et al. developed a Perceiver AR model which had the ability to effectively handle longer sequences with an improved memory efficiency [4]. After listening to the examples from the papers, the generated pieces still consisted of basic harmonic and rhythmic structures and struggled with the complex motifs and harmonic cadences of romantic period piano music. The importance of the research problem not only addresses AI's ability to generate music but also highlights the potential significance of how music could be composed in the future, particularly for those with no previous musical knowledge [5]. The research assumes that neural network models can already be trained to learn the general characteristics and patterns of various musical composers. To help with producing optimal results, the selected MIDI files for the dataset were exact replications of Brahms' piano music without errors and inconsistencies.

Based on the issues raised above, this study focused on understanding what configurations of the neural network models performed best when tasked with producing music in the style of Brahms. Therefore, the research question is:

To what extent can the accuracy of various Neural Network Models, trained with Long Short-Term Memory

and numerous Attention mechanisms, be significantly improved by augmenting MIDI files containing the compositional works of Johannes Brahms with an augmentation pipeline to generate pieces of music that are mistaken by professional musicians, composers and conductors as one of Brahms' own works?

Due to conclusions from studies taken and the general state of knowledge at the time of beginning the project, the null hypothesis for this research project is;

H0: Neural network models cannot generate piano works to the same level of musicality and emotion as Brahms. Due to this, generated pieces will not be statistically similar through music information retrieval or mistaken as a work of Brahms by professional musicians, composers and conductors through a quantitative survey using Likert Scales.

Through the utilisation of an augmentation pipeline to expand the MIDI dataset containing the compositional piano works of Johannes Brahms, more musical variations could be created including transposition, rhythmical and note durations. In addition to this, various preprocessing techniques including track splitting, quantisation and normalisation could help make the MIDI files more readable for the models. This provides an alternate hypothesis;

H1: If an augmentation pipeline is utilised to expand a MIDI dataset of pre-processed files containing the piano works of Johannes Brahms, then various neural network models trained with Long Short-Term Memory and numerous Attention mechanisms could generate pieces of music that is statistically similar to Brahms and could be mistaken as one of Brahms' own piano works by professional musicians, composers and conductors through a quantitative survey using Likert Scales and various Independent-Samples T-Tests and Hotelling's T^2 Tests being implemented to determine whether the p-value is > 0.05 in order to reject the null hypothesis.

This paper contains a total of four sections. Section 2 describes the experiment design, methodology and how the dataset was prepared and pre-processed. Along with this, the neural network models obtained for the project and MIR functions will be explained as well as ethical considerations. Section 3 analyses and evaluates the results of the quantitative experiment and survey to determine if the experiments provide evidence that the null hypothesis is incorrect. Section 4 summarises what has been learnt and proposes recommendations and adjustments for future studies.

II. DESIGN AND METHODOLOGY

The research project was carried out through three stages. Firstly, data was collected, pre-processed, and augmented. The dataset contained 67 MIDI files of Johannes Brahms' piano works and was obtained for offline manipulation from *Classical Archives* and *MIDIworld*. An augmentation pipeline was employed to create variations in melodies, tempo, rhythm, and transpositions. This was followed by obtaining and training various neural network models with the augmented dataset. The models were analysed for training accuracy and loss to determine the best configurations. After training, the best generated examples from each model were analysed through various MIR functions using *MIDIToolbox* and *MIRToolbox* to determine the best performing models [6][7]. Finally, the best models generated pieces of music that were evaluated against the pieces of Brahms' repertoire for statistical equivalence. Along with this, the same generated pieces were used in a quantitative listener survey to test professional musicians, composers and conductors on whether they could differentiate between the generated pieces against the Brahms original. All quantitative experiments involved evaluation to test the research hypothesis through Independent-Samples T-tests and Hotelling's T^2 Tests.

A. Preparation and Preprocessing of Dataset

The preparation and preprocessing of the dataset involved numerous steps to optimise the potential of training the neural network models. Track splitting involved dividing the MIDI tracks into smaller segments of 30 second clips to make the tracks more digestible for the models in training [8]. The conversion of MIDI files into a single track allowed for further simplicity in the files. Normalisation was performed to ensure that the audio levels of the files were at the same amplitude to provide consistent values for training and evaluation tasks. Finally, all the MIDI files were quantized to semiquavers to adjust the timings of notes and align them with the correct timing to ensure consistency in rhythm [9]. An augmentation pipeline was utilised to create variations in melodies, tempo, rhythm and transpositions. Several techniques were used to increase the dataset size. Time-stretching was applied to make each MIDI file 5% faster or slower. Another method was to transpose each of the MIDI files so that the pitches would be raised or lowered by a third [8]. Doing these increased the dataset by 500% with a total of 445 tracks. In comparison with prior studies, data preparation and preprocessing was influenced by previous research, which was collected, adapted and integrated into this paper.

B. Neural Network Models

Numerous Neural Network Architectures were obtained for offline manipulation and trained with the augmented Brahms MIDI dataset. A Recurrent Neural Network was acquired from TensorFlow [9]. A RNN was described by IBM as "a type of artificial neural network which used sequential data or time series data" [10]. Sequential data was utilised to predict the next output based on previous elements in the sequence. RNNs suffer from a vanishing gradient problem. With the neural network using the gradient descent algorithm to update the weight, the gradients therefore decreased in growth the further down the layers the network progressed. A solution to this problem is the use of LSTM which utilises gating mechanisms to

control the movement of information and gradients to allow for the network to learn and maintain information over longer sequences. [11] An LSTM-based RNN was obtained from Huang et al. [12]. Various Transformer models containing different attention mechanisms were obtained from Project Los Angeles [13]. Proposed by Google researchers Vaswani et al., transformer models do not rely on recurring processing of data. Instead, they operate on an attention mechanism [14]. Attention allows for neural networks to concentrate on particular parts of the input. The attention mechanisms tested were:

- Self-Attention – Processes inputs in the same sequence, enabling the model to capture dependencies within the input [15].
- Relative Attention – The relative position of tokens is considered based on the similarity of other tokens in the sequence [16].
- Local Windowed Attention – Restricts attention to a fixed window of tokens, enabling the model to focus on nearby information [17].
- Relative Self-Attention – Combination of Self and Relative attention allowing the model to focus on relevant information based on the positional relationships of tokens [14].
- Sparse Attention – Attends to a subset of tokens instead of the entire sequence to improve efficiency while retaining information [18].

A Perceiver AR model was also obtained from Project Los Angeles. Seen as an improvement to the Transformer model using latent array to distinguish the size of inputs and outputs, allowed for the model to efferently handle longer sequences with an improved memory efficiency [19].

C. Quantitative Experiment & Survey

A quantitative experiment and survey were conducted to test the neural network model's ability to replicate the musical characteristics and motifs of Brahms' piano works. Various MIR variables were utilised to gather various quantitative data from the models to test against not only each other but the original works of Brahms. These variables included:

- Entropy – The measure of uncertainty or unpredictability
- Duration Distribution – The statistical analysis of note durations as well as silence
- Pitch Class Distribution – The evaluation of frequency of musical pitches
- Mean Roughness – The measure of dissonance or clashing sounds
- Global Energy – The total amount of energy held within a waveform
- Normalised Pairwise Variability Index (nPVI) – The analysis of variability between successive durations
- Pulsation Clarity – The strength of the rhythmic pulse in a piece of music

The quantitative survey contained a total of 10 questions all containing the question *"Rate the likelihood that this piece was composed by Johannes Brahms as opposed to being generated by AI"* In random order, 5 pieces contained the works of Brahms and 5 were generated by the models. Answers consisted of a Likert Scale of 5 values ranging from *Definitely generated by AI* to *Definitely generated by Brahms.* Members of the Irish Defence Forces School of Music, RTÉ Concert Orchestra and National Symphony Orchestra were recruited for the survey to provide professional expertise in the subject. Participants were selected based on their extensive experience in classical music, including familiarity with Brahms' works, having performed his pieces in the past.

Using IBM's SPSS software, quantitative values from the MIR functions *MIRToolbox* and *MIDIToolbox* were tested to obtain p-values. Various Independent-Samples T-Tests and Hotelling's T^2 tests were implemented to evaluate musical variables to determine if there was statistical significance between Brahms' piano works and the generated pieces from the AI models.

D. Ethical Considerations

Several ethical considerations were adhered to in order to correctly conduct research including copyright issues and collection of data from survey participants A total of 67 pieces of music were obtained for offline manipulation for the MIDI dataset. According to German Federal Law Gazette, copyright protection for musical and artistic works expired 70 years after the death of the creator. With Brahms passing away in 1897, his compositions therefore resided in the public domain. The use of MIDI files also prevented any issues with performers rights as recordings of Brahms' works were not being used. All the neural network models obtained for testing were open-source and ran under the Apache 2.0 License. MIR tasks were performed using the *MATLAB* functions *MIRToolbox* and *MIDIToolbox*. To utilise the tools, *MATLAB* had to be downloaded free of charge under the GNU General Public License. Ethical considerations were vital when collecting data from personnel for the quantitative survey. Participation in the survey was voluntary and those who chose to partake were informed of the purpose of the study. No personal information was required from participants and the confidentiality of participants was guaranteed from the designer of the survey. The results of the survey were not tampered with and therefore were accurate.

III. EVALUATION

Overfitting issues were observed during training, with the LSTM and Perceiver AR models initially replicating the training material excessively instead of generating unique motifs. To mitigate this, the dataset was further augmented by transposing musical phrases and altering rhythms

through quantisation to introduce additional variations in tempo and phrasing. Experimenting with different temperature values for each model allowed for control over the balance between simplicity and randomisation. With lower temperatures producing more basic outputs that resembled the training data, while higher temperatures encouraged greater diversity but could lead to compositions with less structure.

The model performance metrics stated that the transformer model with relative self-attention scored the best training loss and accuracy with scores of 0.015 and 0.995 respectively. The recurrent neural network scored worst in terms of training accuracy with a score of 0.628.

Through quantitative experimentation and a survey, the neural network models utilised for the project were evaluated extensively in a numerical form and through professional human judgement in order to confirm or refute the research hypothesis that neural network models could generate pieces of music with statistically similar musical characteristics and emotion as Brahms. To conduct a fair experiment, each model had to generate a two-minute-long piece which contained 300 prime tokens (30 seconds) from the beginning of six of Brahms' piano works. These generated pieces were then compared with the first two minutes of the original Brahms piece to evaluate the evolution of the generated pieces and determine their ability to maintain the style and structure of Brahms. MIR evaluation concluded that the Transformer models with self-attention, local windowed attention and relative global attention performed best in generating music most similar to the Brahms original.

Several statistical tests were conducted on the best performing models to obtain p-values to test the research hypothesis. The variables Entropy, nPVI, Global Energy, Mean Roughness and Pulsation Clarity were tested with an Independent-Samples t-test and the variables Duration Distribution and Pitch Class Distribution were tested with a Hotelling's T^2 test. Testing concluded that no statistical significance was found with most of the variables therefore supporting the alternative hypothesis that neural network models can produce music similar to Brahms. However, the variables Entropy and Global Energy were deemed to contain statistical significance within them stating that further work must be done to improve complexity, uncertainty and energy to a similar level to Brahms. Using *MIRToolbox*, the waveforms of the generated pieces were evaluated.

Figure 1 shows the brightness curve of Brahms' 2 Rhapsodies No. 1 alongside the generated piece from the transformer model with local windowed attention. The generated piece displays a greater variance of frequencies, resulting in a higher entropy score.

The difference in global energy is depicted in Figure 2 where the temporal evolution curve reveals a much greater variance in the generated piece compared to Brahms' original work. While Brahms' piece maintains a steady flow of increase and decrease of tension, the generated piece has much greater variations in timbre and harmonics throughout. The evaluation of these waveforms determined that the entropy and global energy values may have been much higher than Brahms' works due to the *MuseNet* inspired workflow the transformer models undertook causing the pieces to be generated in blocks and therefore sound unnatural.

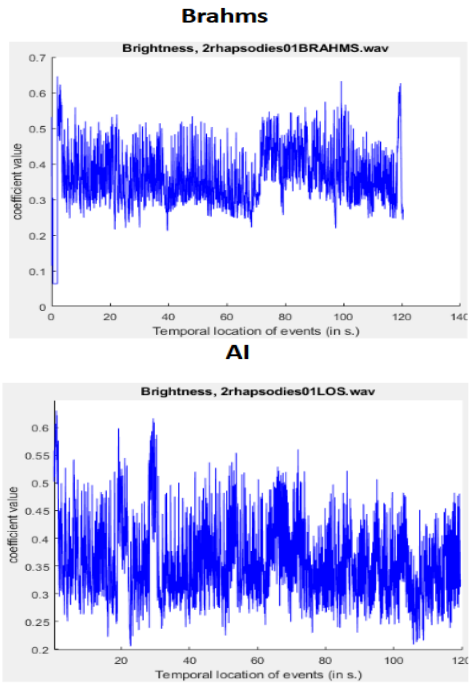


Figure 1. Brightness Curve between Brahms and AI

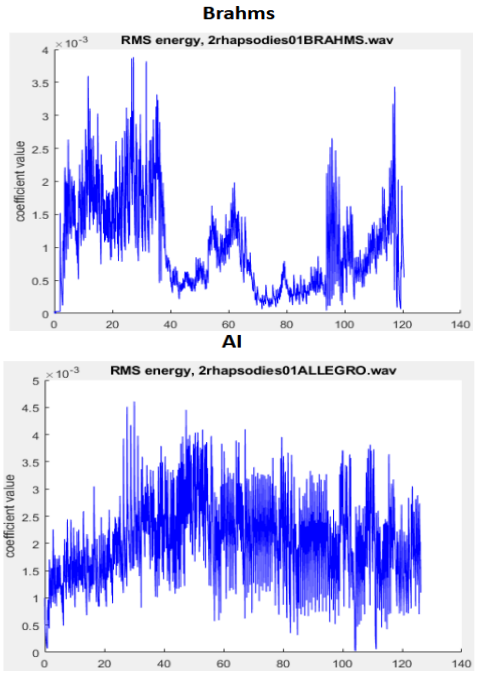


Figure 2. Temporal Evolution curves for Brahms and AI

Figure 3 displays the Pitch Class Distribution in the form of a box plot for Brahms' *3 Intermezzi No.1* and the generated piece from the transformer model with local windowed attention. With the piece being in the key of D# Major, which has an enharmonic equivalent of C minor, in order to stay within the key signature, the majority of the notes must be from the following triads:

- D# Major – D#, G, A#
- C Minor – C, D#, G

The pitch class distribution showed that both the original Brahms piece and AI generated were kept within these guidelines, with particular emphasis being placed on the notes D# and G as they feature in the triads of both D# major and C minor. Although the transformer model focused on the notes within the two triads to ensure consistency in the key signature, it was apparent that the model was reluctant to incorporate accidentals to further add colour to the piece. The use of extended chords was a key factor of the romantic period in which Brahms lived in and it was an era that bridged the gap between classical and modern music. *MIRToolbox* was able to identify both the Brahms and AI pieces to be in the key of D# major. With the function *mirmode,* it identified that the generated piece scored a higher probability of being in a major key than the Brahms original. Although it was positive that the generated piece was able to keep within the key signature for longer generated sequences, it does show an inability to evolve melodically into different harmonics.

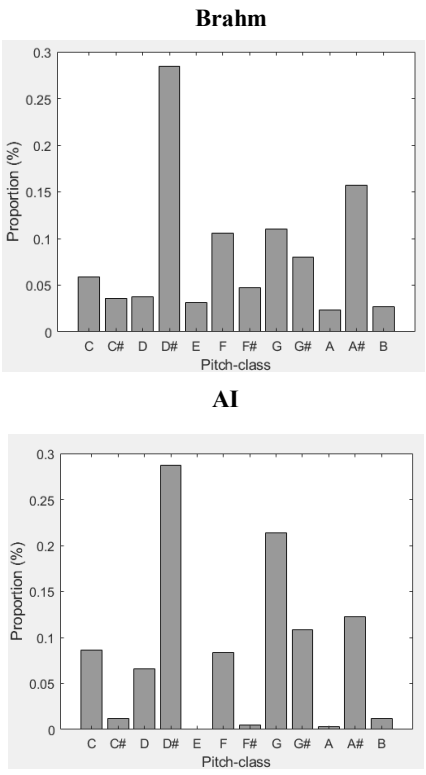


Fig. 3 Box Plot of the Pitch Class Distribution between Brahms and AI

A quantitative survey was also conducted to obtain human evaluation on 30 second clips of generated pieces from the models against the original works of Brahms. A total of 56 participants featuring only professional musicians, composers and conductors, displayed difficulty in recognising the distinction between the Brahms and generated pieces provided, with the majority incorrectly identifying two of the generated pieces as one of Brahms' own works.

Table 1 shows the percentages for each of the questions alongside whether the track was that of Brahms or AI. The total score was also calculated with the number of responses per answer being multiplied dependant on how similar to Brahms it was scored with *Definetly generated by AI* scoring 1 and *Definitely generated by Brahms* scoring 5. This was designed so that uncertainty was treated as a reward for the AI models as they still had not been identified as not Brahms.

An Independent-Samples T-Test was conducted which stated that there was no statistical significance between the Brahms and generated pieces therefore supporting the alternative hypothesis that neural network models trained with an augmented and pre-processed dataset could generate music at the level of musicality and emotion as of Brahms so much that through a quantitative survey the difference could not be identified by professional musicians, composers and conductors.

TABLE 1. QUANTITATIVE SURVEY RESULTS

	AI	Probably AI	Unsure	Probably Brahms	Brahms	Total Score
Brahms	3.64%	5.45%	3.64%	36.36%	**50.91%**	**237**
Brahms	12.50%	25.00%	3.57%	**37.50%**	21.43%	**185**
Brahms	0%	23.21%	7.14%	**39.29%**	30.36%	**211**
Brahms	17.86%	**44.64%**	3.57%	28.57%	5.36%	**145**
Brahms	7.27%	7.27%	18.18%	**41.82%**	25.45%	**207**
AI	23.21%	**32.14%**	12.50%	23.21%	8.93%	**147**
AI	10.71%	19.64%	19.64%	**28.57%**	21.43%	**185**
AI	5.36%	**41.07%**	14.29%	32.14%	7.14%	**165**
AI	14.55%	**34.55%**	5.45%	32.73%	12.73%	**165**
AI	7.27%	30.91%	7.27%	**36.36%**	18.18%	**183**

IV. CONCLUSION AND FUTURE WORK

The work conducted in this paper differed to previous studies as it trained various neural network models with a dataset containing the piano works of Brahms, an important figure of the Romantic Period in Classical Music. By doing this, a gap in the research was addressed by analysing a very important composer in an era of classical music were harmonic and rhythmic structures began to diverge from the traditional aspects of Renaissance and Classical Period music while also bridging the gap between traditionalism and modernism. While the literature review stated a gap in the research that previous models struggled with complex motifs and harmonic cadences of romantic period piano music. This paper found that statistical testing in various

musical categories stated there was not statistical significance between the Brahms and AI generated pieces. A quantitative survey containing participants who were educated in the subject mistook two of the neural network models generated pieces as Brahms' own works suggesting the model's ability to generate pieces of music with the complexity in rhythmic and harmonic characteristics of Brahms.

The results from the research carried out suggest that transformer models with self-attention, relative self-attention and local windowed attention were able to generate various characteristics to a statistically similar level to Brahms with a dataset of his piano works by utilising an augmentation pipeline and various preprocessing techniques. A couple of musical characteristics however proved to be statistically significant to Brahms, these being entropy and global energy. This concludes that while the transformer models are able to replicate a vast amount of Brahms' compositional traits, they still fall behind in reproducing the rhythmical and harmonical complexities, uncertainties and global energy levels of Brahms' works. The possibility of increasing the dataset to the orchestral, ensemble and choral works of Brahms could greatly increase the abilities of generated music from just solo piano works. This also could adhere to limitations regarding a small dataset and therefore improve accuracies in entropy and global energy from the generated pieces.

While participants noted difficulty in distinguishing between the Brahms and AI pieces, some commented that the use of MIDI files made all the music sound robotic and therefore made it even more challenging to differentiate between the two. Future work could focus on converting the generated pieces into musical notation and having a professional pianist perform them. This would enable an experiment to assess the generated music on an acoustic piano, the instrument it was originally intended for.

REFERENCES

[1] J. D. Fernández and F. Vico, "AI Methods in Algorithmic Composition: A Comprehensive Survey," *Journal of Artificial Intelligence Research*, no. 48, pp. 513-582, 2013. https://doi.org/10.48550/arXiv.1402.0585

[2] K. Zheng, R. Meng, C. Zheng, X. Li, J. Sang, J. Cai. and J. Wang, "EmotionBox: a music-element-driven emotional music generation system using Recurrent Neural Network," *Frontiers in Psychology*, no. 13, pp. 1-14, 2021. https://doi.org/10.3389/fpsyg.2022.841926

[3] R. Child, S. Gray, A. Radford and I. Sutskever, "Generating Long Sequences with Sparse Transformers. *PsyArXiv,* no. 1, pp. 1-10, 2019. https://doi.org/10.48550/arXiv.1904.10509

[4] C. Hawthorne, A. Jaegle, C. Cangea, S. Borgeaud, C. Nash, M. Malinowski, S. Dieleman, O. Vinyals, M. Botvinick, I. Simon, H. Sheahan, N. Zeghidour, J. B. Alayrac, J. Carreira, and J. Engel, "General-purpose, long-context autoregressive modelling with Perceiver AR," *In Proc of the International Conference on Machine Learning*, no. 39, pp. 1-24, 2022. https://doi.org/10.48550/arXiv.2202.07765

[5] E. Deruty, M. Grachten, S. Lattner, J. Nistal and C. Aouameur, "On the Development and Practice of AI technology for Contemporary Popular Music Production," *Transactions of the International Society for Music Information Retrieval,* no. 5, pp. 35-49, 2022. https://doi.org/10.5334/tismir.100

[6] O. Lartillot, P. Toiviainen, P. Saari and T. Eerola, "MIRtoolbox," 2007. [Online]. Available: https://www.jyu.fi/hytk/fi/laitokset/mutku/en/research/materials/mirtoolbox

[7] T. Eerola and P. Toiviainen, "MIDI Toolbox: MATLAB Tools for Music Research," 2004. [Online]. Available: https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=6e06906ca1ba0bf0ac8f2cb1a929f3be95eeadfa

[8] S. Oore, I. Simon, S. Dieleman, D. Eck and K. Simonyan, "This Time With Feeling: Learning Expressive Musical Performance," *Neural Computing and Applications,* no. 32, pp. 995-967, 2018. https://doi.org/10.1007/s00521-018-3758-9

[9] J. A. Franklin, "Jazz Melody Generation from Recurrent Network Learning of Several Human Melodies," *In Proc of the International Florida Artificial Intelligence Research Society Conference,* no. 18, pp.57-62, 2005. https://cdn.aaai.org/FLAIRS/2005/Flairs05-010.pdf

[10] TensorFlow, "Generate music with an RNN," n.d. [Online]. Available: https://www.tensorflow.org/tutorials/audio/music_generation

[11] IBM, "What are recurrent neural networks?" n.d. [Online] Available: https://ibm.com/topics/recurrent-neural-networks

[12] S. Hochreiter and J. Schmidhuber, "Long Short-Term Memory," *Neural Computation,* no. 9, pp. 1735-1780, 1997. https://doi.org/10.1162/neco.1997.9.8.1735J.

[13] C. Z. A. Huang, A. Vaswani, J. Uszkoreit, N. Shazeer, I. Simon, C. Hawthorne, A. M. Dai, M. D. Hoffman, M. Dinculescu and D. Eck, "Music Transformer: Generating Music with Long-Term Structure," *Proceedings of the International Conference on Learning Representations,* no. 17, pp. 1-14, 2018. https://doi.org/10.48550/arXiv.1809.04281

[14] A. Sigalov, "Project Los Angeles" 2019. [Online]. Available: https://github.com/asigalov61

[15] A. K. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser and I. Polosukhin, "Attention is All you Need," *Advances in Neural Information Processing Systems,* no. 31, pp. 1-15, 2017. https://doi.org/10.48550/arXiv.1706.03762

[16] A. K. Huang, P.A. Szerlip, M. E. Norton, T.A. Brindle, Z. Merritt and K.O. Stanley, "Visualizing music self-attention" *In Proc of the Conference on Neural Information Processing Systems,* no.32, pp. 1-5, 2018. https://openreview.net/pdf?id=ryfxVNEajm

[17] P. Shaw, J. Uszkoreit and A. Vaswani, "Self-Attention with Relative Positional Representations," *In Proc of the 2018 Conference of the North American Chaper of the Association for Computational Linguistics: Human Language Technologies,* no.2, pp. 464-468, 2018. https://doi.org/10.18653/v1/N18-2074

[18] Z. Liu, Y. Lin, H. Hu, Y. Wei, Z. Zhang, S. Lin and B. Guo, "Swim Transformer: Hierarchical Vision Transformer using Shifted Windows," *In Proc of the IEEE/CVF International Conference on Computer Vision,* no.23, pp. 9992-10002, 2021. https://doi.org/10.48550/arXiv.2103.14030

[19] A. Jaegle, F. Gimeno, A. Brock, A. Zisserman, O. Vinyals and J. Carreira, "Perceiver: General Perception with Iterative Attention," *In Proc of the International Conference on Machine Learning,* no. 38, pp. 1-43, 2021. https://doi.org/10.48550/arXiv.2103.03206

Practical Applications of State-Of-The-Art Large Language Models to Solve Real-World Software Engineering Problems Autonomously

Yurij Mikhalevich
QA Wolf
Dubai, United Arab Emirates
email: yurij@mikhalevi.ch

***Abstract*—This paper researches the application of state-of-the-art large language models to autonomously solve real-world software engineering problems based on the problem description intended for humans. For this research, we picked 10 outstanding GitHub issues of different difficulty levels in the Aibyss project. We tasked an AI agent to autonomously solve them based solely on the GitHub Issue description intended for human software engineers. As part of this research, we compared the following large language models: Claude Sonnet 3.7, DeepSeek-V3, DeepSeek-R1, and o3-mini-high. We used the Aider agent to solve the problems. Additionally, we have evaluated the Claude Code agent as one of the best closed-source AI software engineering agents. We have found that the best performance is achieved by Claude Sonnet 3.7 with reasoning enabled – with the Aider agent and the Claude Code agent. Both of them provided working solutions to 5 out of 10 GitHub issues. We analyze the agents' behaviors, including reasoning steps, common failure modes, and the impact of reasoning tokens. The results highlight both the promise and the current limitations of autonomous LLM-based software engineering.**

Keywords-code generation; large language models; AI agents; natural language processing

I. INTRODUCTION

Recent advances in large language models (LLMs) have led to powerful code generation systems capable of assisting software developers. Models like o3-mini and DeepSeek-R1 can translate natural language specifications into code with impressive accuracy [1][2]. These models underpin tools such as GitHub Copilot and Cursor Composer, which have been rapidly adopted as "AI pair programmers" to autocomplete code and suggest solutions [3][4][5]. Studies show that such tools can improve developer productivity, but also raise questions about reliability and how developers interact with AI-generated code. So far, these AI coding assistants operate with a human in the loop: the developer guides the process, reviews suggestions, and tests or debugs the outcomes.

A growing area of interest is whether state-of-the-art LLMs can operate more *autonomously* to tackle software engineering tasks end-to-end. Inspired by agentic frameworks like ReAct [6] and the popularity of systems such as AutoGPT [7], researchers have begun treating LLMs as independent problem-solvers rather than just interactive assistants. For example, the ReAct paradigm by Yao et al. [6] enables an LLM to generate reasoning traces and act on them iteratively, and projects like AutoGPT aim to let an LLM plan and execute a sequence of steps towards a high-level objective. Meanwhile, multi-agent approaches have emerged to coordinate multiple LLMs or tools in specialized roles (e.g., planning vs. coding) to solve complex tasks [8]. Existing benchmarks of these autonomous LLM agents indicate that the choice of the underlying model has a critical impact on success: for instance, GPT-4 can substantially outperform GPT-3.5 or smaller models in autonomous decision-making tasks [9].

Recent advances in LLM-based agents open up the possibility of automating several steps in the classical software development lifecycle. Tasks such as requirement interpretation, code generation, test creation, and documentation can now be at least partially handled by these agents. Especially in the early phases-like prototyping or resolving isolated issues from natural language descriptions-LLMs show a strong capacity for autonomous operation [10]. However, phases involving architectural decisions, integration testing, and final validation still rely heavily on human expertise and oversight.

Despite the growing capabilities of these models, transitioning from generated code to a trusted, production-grade system presents significant challenges. These include ensuring correctness, robustness, maintainability, and compliance with domain-specific standards. Generated code often lacks integration context, can contain subtle bugs, or may not align with broader system constraints. Therefore, human-in-the-loop review, continuous integration pipelines, and formal verification methods are often critical to close the gap between raw LLM output and trustworthy software.

Research in software configuration management (SCM) is increasingly intersecting with AI-driven development. Some ongoing work investigates how agents can update deployment configurations, manage dependencies, and track version history intelligently. Emerging tools explore LLMs not just as code generators but as collaborative participants in the evolution of codebases, integrating with version control systems and automating routine deployment and maintenance tasks [11].

Moreover, there is a growing interest in fine-tuning or pretraining LLMs for specialized tasks such as software update management. These niche LLMs aim to support activities like patch generation, changelog summarization, and semantic versioning analysis. This area is attracting attention as organizations seek to reduce the overhead of continuous software maintenance through domain-adapted language models [10][12].

In this work, we explore the practical application of cutting-edge LLMs as autonomous software engineers on real-world tasks. We design an experiment in which an AI-driven coding agent is given only the natural-language description of a software issue (as one would find in a bug tracker or feature request) and is tasked with resolving the issue by modifying

the codebase, without human assistance. We evaluate the following state-of-the-art LLMs in this autonomous setting: Claude Sonnet 3.7 [13], DeepSeek-V3 [14], DeepSeek-R1 [2], and o3-mini-high [1]. We have used the Aider [15] agent to solve the problems – one of the best open-source AI software engineering agents. Additionally, we have evaluated the Claude Code [13] agent as one of the best closed-source AI software engineering agents. We examine not only whether the LLM-powered agent can produce a working solution, but also the quality of the solution (linting, code style, user experience) and the computational cost (API calls/tokens consumed).

The results of the research show that the best performance is achieved by Claude Sonnet 3.7 with reasoning enabled – with both the Aider agent and the Claude Code agent. Both of them provided working solutions to 5 out of 10 GitHub issues. Surprisingly, Aider paired with o3-mini-high performed the worst out of all the agents and has shown the worst understanding of the problems.

The rest of the paper is organized as follows. Section 2 provides an overview of the related works. Section 3 describes the method used in the research. Section 4 presents the experiment implementation details. Section 5 presents the evaluation results. Section 6 presents a detailed analysis of agent behaviors and failure modes. Finally, Section 7 concludes the paper.

II. Related Works

In this section, we examine the existing literature and research efforts that form the foundation for our current study. Prior investigations have established several key approaches and methodologies that inform our work.

A. LLMs for Code Generation and Assistance

The use of large neural models for code generation has rapidly progressed in recent years. OpenAI's Codex model [16], which powers GitHub Copilot, was among the first to demonstrate that an LLM trained on vast amounts of code can produce syntactically correct and often functionally correct code for given descriptions. Subsequent models have pushed these capabilities further: DeepMind's AlphaCode achieved a performance on par with average human competitors in programming contests[17], signaling the potential of LLMs to handle complex algorithmic problems.

Recent developments in the field have demonstrated significant progress in computational capabilities. Specifically, models such as OpenAI 03-mini-high, DeepSeek-R1, and Claude Sonnet 3.7 have established new performance standards [1][13]. These advancements indicate the continued rapid evolution of LLM capabilities, with potential implications for fully autonomous software engineering agents.

B. LLM-Based Autonomous Agents

Beyond single-turn code completion, the idea of an LLM-driven agent that can perform multi-step tasks has gained traction. The ReAct framework [6] pioneered the combination of *chain-of-thought* reasoning [18] with action execution, enabling an LLM to decide not only *what to think next* but also *what action to take* in a unified prompting strategy. This idea of using the LLM's own output as an intermediate state has influenced many subsequent systems.

In early 2023, a series of autonomous agent prototypes built on GPT-4 (such as AutoGPT) captured popular imagination. These systems prompt the LLM to continuously plan and execute sub-tasks towards a given objective, simulating an "AI agent" that can function without user prompts for each step.

Recent developments in AI-powered code generation have witnessed significant breakthroughs with the emergence of sophisticated agent-based systems. Notably, Courser Composer and Aider have demonstrated improved capabilities in autonomous programming tasks [5][15].

Our work can be seen as an instance of an LLM agent applied to a focused real-world task: given a specific issue in an existing software project, the agent (backed by an LLM) must understand the problem, read and modify the project's code, and submit a solution. We contribute new data on how today's strongest LLMs perform in this autonomous coding scenario, complementing prior research.

III. Method

Our research methodology is designed to evaluate each LLM's ability to autonomously resolve real software issues under controlled conditions. We selected the open-source project Aibyss, a web-based AI competition game, as our testbed. Aibyss is a TypeScript project (Nuxt/Vue frontend with a Node.js backend using Prisma ORM) where users write AI bots to compete in a game [19]. We chose Aibyss because it is a non-trivial codebase with realistic features and bugs, yet manageable in size. From Aibyss's issue tracker, we picked ten issues that were open and well-described. These issues covered a range of feature requests and bug fixes and were labeled by us based on the perceived difficulty as "easy," "medium," or "harder".

A. Task Selection

The 10 issues included 3 labeled "easy", 4 "medium", and 3 "harder". Each issue consisted of a title and a description intended for human developers. We did not provide any additional hints or test cases to the agent beyond this text.

Below is the complete list of problems that we selected and their issue titles, presented as-is:

1) easy - "feat: draggable splitter between the code and the game screen should remember its position between the page reloads"
2) easy - "feat(rating): highlight top results in k/d, kills, deaths, and food eaten columns in the rating table"
3) easy - "chore(World): double the frequency of food spawns"
4) medium - "feat: allow turning off the bots of some users by setting the "inactive" field in the database on the user object to 'true"'
5) medium - "feat: ensure that the game screen occupies all available free space to the right of the code editor"

6) medium - “feat(rating): add a new column to the rating table displaying the number of times the user submitted the code”
7) medium - “feat(sandbox): add an option to turn off sprites and replace them with circles to make debugging easier”
8) harder - “bug: fix the issue causing the bot code to submit when the user opens “API reference”
9) harder - “feat: add code versions and an option to revert to a previous version”
10) harder - “feat: surface bot execution errors to the user”

The actual GitHub issues with their descriptions can be found on the Aibyss project GitHub issues page [20].

B. Agents and LLM Variants

We evaluated six agent configurations:

- Aider 0.75.2 + o3-mini-high 2025-01-31
- Aider 0.75.2 + DeepSeek-V3
- Aider 0.75.2 + DeepSeek-R1
- Aider 0.75.2 + Claude Sonnet 3.7 20250219
- Aider 0.78.0 + Claude Sonnet 3.7 20250219 with 32k thinking tokens – in this variant, we enabled the “thinking mode” in Aider (using v0.78.0 with thinking support for Claude 3.7)
- Claude Code 0.2.35 – Anthropic’s Claude Code is a proprietary agent with a CLI interface very similar to Aider’s that uses the Claude Sonnet 3.7 model under the hood; this can be seen as a closed-source counterpart to Aider, specifically tuned for Claude [13]

C. Autonomy and Stopping Criteria

We configured the agents to operate fully autonomously. Aider was run with the `--yes-always` flag, meaning it would automatically apply its proposed actions. In the case of Claude Code, we approved all its prompts manually. Each agent was allowed to iterate until it produced no further actions.

One exception to full autonomy was with the o3-mini-high model in Aider: often this model did not automatically load files it needed, and would ask the user to add certain files to its context. Whenever *Aider+o3-mini-high* requested a file, we manually added exactly that file (and no additional help), then let it continue. No other agent required such interventions.

D. Evaluation Criteria

After each agent run, we collected the resulting code changes (if any) and deployed/tested the application to judge success. We evaluated outcomes on several criteria:

- **Works (Yes/No):** Did the changes address the issue from the end-user’s perspective? For a feature request, this meant the new functionality works as intended. For a bug, the erroneous behavior was fixed.
- **Linting Check Pass:** We ran the project’s linting scripts. If the agent’ss final code did not pass them, we marked that as a quality issue.
- **User Experience (UX):** We manually inspected if the solution introduced any noticeable UX problems (e.g., a feature works but has a confusing UI or performance lag).
- **Code Quality:** We reviewed the diffs to assess if the solution was implemented in a reasonable and maintainable way. Inefficiencies, unmaintainable code, and obvious bugs in the implementation were noted.

We selected these criteria because they mirror how work performed by a human software engineer is usually evaluated. These qualitative judgments were used to label each successful solution with additional notes (e.g., “works, but suboptimal code” or “works, except fails linting”). Finally, we measured the cost of each solution in USD.

IV. Implementation Details

All agent runs were conducted in a consistent environment. We created a fresh Docker container for each run, which checked out the Aibyss repository at commit `b4e58b2` (to ensure all models started from identical code) and installed the necessary tools (Node.js, Aider, Claude Code, etc.). The agent was then launched inside the container and given the issue text to solve. The prompt given to each agent was uniform: `"Please solve the following issue. Title: <issue title> Description: <issue body>"`. We ensured the project’s dependencies and database (SQLite for this test) were properly set up in each container so that the agent could run the app or tests if it chose to. The Aibyss codebase was about 3.5k lines of TypeScript/JavaScript. Each agent configuration was run on each of the 10 issues, yielding 60 trials in total.

After an agent completed, we committed its changes to a new branch and opened a pull request on GitHub. This allowed us to use continuous integration (CI) results as an additional datapoint. We then manually reviewed and tested the branch as described in the evaluation criteria. All of the PRs created as part of this research can be found on GitHub [21].

V. Results

Table I summarizes the performance of each agent configuration across the 10 issues. We report for each issue whether the agent produced a working solution, along with notes on linting, UX, and code quality. We also report the approximate API cost incurred for that issue’s attempt. A “doesn’t work” or “it didn’t understand the problem” indicates the agent failed to solve the issue.

Looking at the overall success rates (“solved problems” in the Total row), we see a clear ranking of the models. The Claude Sonnet 3.7 with reasoning (both with Aider and Claude Code) solved 5 out of 10 issues, the highest of any configuration. In contrast, DeepSeek-V3 and -R1 solved 2 each, and the o3-mini model solved only 1. The standard *Aider+Claude* (with no reasoning) solved 3. Enabling Claude to use “thinking” (32k tokens context for chain-of-thought) allowed it to solve two additional issues that it failed with a shorter context, showing that reasoning improved performance.

TABLE I
AGENT EVALUATION RESULTS

Problem	Aider 0.75.2 + o3-mini-high 2025-01-31	Aider 0.75.2 + DeepSeek-V3	Aider 0.75.2 + DeepSeek-R1	Aider 0.75.2 + Claude Sonnet 3.7 20250219	Aider 0.78.0 + Claude Sonnet 3.7 20250219 with 32k thinking tokens	Claude Code 0.2.35
1	cost: $0.04 doesn't work	cost: $0.0046 doesn't work	cost: $0.0092 doesn't work	cost: $0.12 ✓**works** linter check fail UX is bad code is bad	cost: $0.20 ✓**works** linter check fail UX is bad ✓**code is good**	cost: $0.2928 ✓**works** ✓**linter check pass** UX is bad ✓**code is good**
2	cost: $0.05 it didn't understand the problem	cost: $0.0037 doesn't work	cost: $0.0070 ✓**works** linter check fail ✓**UX is good** ✓**code is good**	cost: $0.04 doesn't work	cost: $0.07 ✓**works** linter check fail ✓**UX is good** ✓**code is good**	cost: $0.1175 doesn't work
3	cost: $0.03 ✓**works** ✓**linter check pass** ✓**UX is good** ✓**code is good**	cost: $0.0033 ✓**works** ✓**linter check pass** ✓**UX is good** ✓**code is good**	cost: $0.0066 ✓**works** linter check fail ✓**UX is good** ✓**code is good**	cost: $0.04 ✓**works** ✓**linter check pass** ✓**UX is good** ✓**code is good**	cost: $0.07 ✓**works** ✓**linter check pass** ✓**UX is good** ✓**code is good**	cost: $0.1151 ✓**works** ✓**linter check pass** ✓**UX is good** ✓**code is good**
4	cost: $0.07 doesn't work	cost: $0.0043 doesn't work	cost: $0.0070 doesn't work	cost: $0.06 doesn't work	cost: $0.08 doesn't work	cost: $0.4942 doesn't work
5	cost: $0.04 doesn't work	cost: $0.0042 doesn't work	cost: $0.0079 doesn't work	cost: $0.07 doesn't work	cost: $0.08 doesn't work	cost: $0.2085 doesn't work
6	cost: $0.07 doesn't work	cost: $0.0046 it didn't understand the problem	cost: $0.0092 doesn't work	cost: $0.07 it didn't understand the problem	cost: $0.10 it didn't understand the problem	cost: $0.2523 it didn't understand the problem
7	cost: $0.22 doesn't work	cost: $0.0090 doesn't work	cost: $0.02 doesn't work	cost: $0.06 doesn't work	cost: $0.09 ✓**works** linter check fail ✓**UX is good** code is bad	cost: $0.4650 ✓**works** linter check fail minor UX issues code is bad
8	cost: $0.09 doesn't work	cost: $0.0031 ✓**works** ✓**linter check pass** ✓**UX is good** code is bad	cost: $0.0063 doesn't work	cost: $0.04 ✓**works** linter check fail ✓**UX is good** code is bad	cost: $0.07 ✓**works** linter check fail ✓**UX is good** code is bad	cost: $0.1518 ✓**works** ✓**linter check pass** ✓**UX is good** ✓**code is good**
9	cost: $0.08 doesn't work	cost: $0.0062 doesn't work	cost: $0.01 doesn't work	cost: $0.10 doesn't work	cost: $0.12 doesn't work	cost: $0.53 ✓**works** linter check fail UX issues code is bad
10	cost: $0.07 doesn't work	cost: $0.0082 doesn't work	cost: $0.02 doesn't work	cost: $0.05 doesn't work	cost: $0.16 doesn't work	cost: $0.50 doesn't work
Total	cost: $0.76 1/10 solved	**cost: $0.05** 2/10 solved	cost: $0.10 2/10 solved	cost: $0.65 3/10 solved	cost: $1.04 **5/10 solved**	cost: $3.13 **5/10 solved**
Easy solved	1/3	1/3	2/3	2/3	**3/3**	2/3
Medium solved	0/4	0/4	0/4	0/4	**1/4**	**1/4**
Harder solved	0/3	1/3	0/3	1/3	1/3	**2/3**

In terms of difficulty, all agents found the easy issues more approachable: *Aider+Claude 3.7 with thinking* solved all 3 easy tasks, and even the weakest model solved one easy issue. The medium tasks proved challenging: only the Claude Sonnet 3.7 with reasoning (both with Aider and Claude Code) managed to solve 1 out of 4 medium issues. Harder tasks (8, 9, 10) saw partial success: Claude Code solved two (Issues 8 and 9), while *Aider+Claude 3.7 with thinking* and *Aider+DeepSeek-V3* each solved one (Issue 8). No model could handle Issue 10, a complex feature involving tracking and displaying bot errors.

We also observe notable differences in solution quality. For instance, Issue 3 was solved by almost all models, but DeepSeek-R1's solution failed linting due to minor format issues, whereas others passed. In Issue 1, all *Aider+Claude* solutions worked functionally, but they left some UX issues (the page loaded with a flicker in the splitter position) and non-ideal code; the Claude Code agent's solution was slightly better (fixing the linting problem) than *Aider+Claude*'s. The other models did not even reach a working solution for Issue 1. Another example is Issue 6. In Issue 6, none of the agents solved it and most (even Claude) misunderstood the requirement – they implemented a total count instead of a 7-day count for a metric. This shows that LLMs can misread context that a human developer is expected to catch. Additional context or clarification in prompts might be needed for such cases.

Cost-wise, the DeepSeek-V3 and DeepSeek-R1 are very cheap to run totalling $0.05 and $0.10 USD per 10 solutions produced respectfully. While using Claude Sonnet 3.7 was

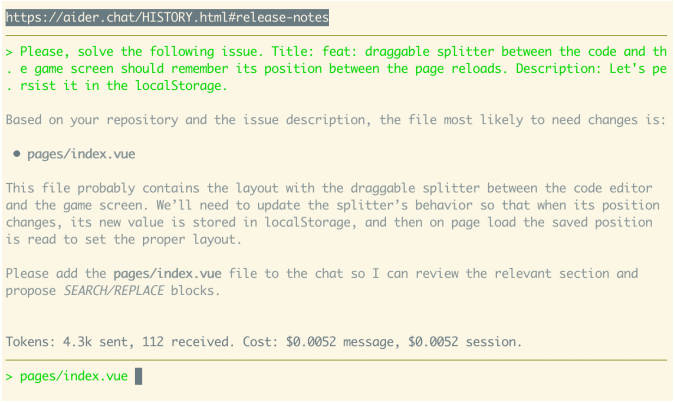


Figure 1. *Aider+o3-mini-high* prompting the user to manually add the file it needs

more expensive, these costs were fairly low in absolute terms (each issue well under $1 for Claude). Notably, Claude Code consumed roughly 3× the tokens of *Aider+Claude*, meaning it took more steps or context per issue. This aligns with Claude Code's behavior of running tests and iterating on a solution autonomously. *Aider+Claude* with reasoning achieved the same 5/10 solves at one-third the cost of Claude Code. In a scenario where API cost is a concern, the improvement in solution quality by Claude Code may not currently justify its higher cost.

VI. Discussion

To better understand the experimental outcomes, we performed a qualitative analysis of each agent's behavior and the solutions (or attempts) they produced. Here we discuss key insights and failure modes.

A. Need for Context Awareness

One striking limitation was observed with o3-mini-high. This agent often failed to load relevant files autonomously. In multiple issues it would stop and ask for a file (for example, it did not automatically open the file containing a function mentioned in the issue). We had to manually provide the file for it to proceed. This behavior is depicted in Figure 1, which shows a screenshot of the *Aider+o3-mini* agent prompting the user to add a file to the context. Notably, none of the other models exhibited this limitation — they have added needed files to the context via the Aider agent's capabilities. In addition to this, the *Aider+o3-mini-high* demonstrated lack of context awareness that significantly impeded its performance. Even after adding files, its solutions were usually incomplete or incorrect.

B. Common Reasoning Errors

Several agents made similar mistakes on certain issues, suggesting the problem might lie in the prompt or the environment rather than idiosyncrasies of one model. For example, in Issue 4 (a medium bug fix involving adding a TypeScript module import), the DeepSeek and o3-mini-high models all attempted to import a database object incorrectly (they used a wrong path), possibly picking up a pattern from elsewhere in code. This parallel behavior implies that the initial system might have led the models down a similar path, or they all latently "agreed" on a plausible but wrong solution. It highlights that autonomous agents might need better guardrails or self-checks for such predictable pitfalls. A possible remedy could be incorporating static analysis: e.g., after a code edit, have the agent verify imports or run a quick compile step (which Claude Code was usually doing).

Another pervasive issue was misunderstanding of requirements. Issue 6 (add a weekly count to the rating table) was misinterpreted by every model as a total count. This happened because the issue title was somewhat ambiguous and the description didn't explicitly mention the 7-day window (it relied on context that all other stats in that table were weekly). Our LLMs did not infer that context. This kind of mistake is hard for an agent to catch without more project knowledge. It suggests that for certain tasks, an autonomous agent might benefit from a mechanism to ask clarifying questions – something we did not allow in this study.

C. Behavior of Claude Code vs. Aider

Using the same model (Claude Sonnet 3.7), the Claude Code agent and the Aider agent exhibited different styles. Claude Code was much more thorough: it would run the project's tests when available, and even run the linter, essentially simulating what a careful developer would do. Claude Code was also the only agent that was actually creating the Prisma database migrations in addition to changing the schema. The downside was that Claude Code consumed more time and tokens.

D. Successful Case Study (Issue 7 - Sandbox Sprites Toggle)

This was a medium difficulty feature that only the Claude-based agents solved. The task was to add a user option to replace graphical sprites with simple circles in the game (for debugging). The *Aider+Claude with reasoning* agent produced a clean solution: it introduced a button to switch to and from the debug mode and a corresponding logic for the toggle. The implementation was not trivial – it required understanding how the rendering loop worked. The other models failed here likely because they got confused by the rendering code. *Aider+Claude with reasoning*'s success in Issue 7 demonstrates the benefit of reasoning. Interestingly, Claude Code's working solution for the same issue was a bit messy (the circle drawing code is unnecessary complex and is repeated in two different places). This suggests that while Claude Code's strategy of iterative refinement is helpful, it doesn't guarantee a better solution design.

E. Partial Success and Limitations (Issue 9 - Code Versioning)

One of the "harder" issues (Issue 9) was to implement code version tracking and allow reverting to previous versions. Claude Code was the only agent to achieve a working solution here. It modified the bot code storing logic to store versions, added a new API to list code versions, built a frontend component to list versions, implemented logic to restore the older

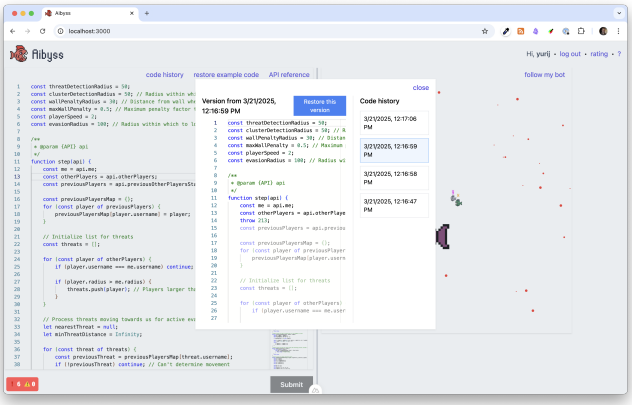


Figure 2. UI added by Claude Code for code versioning (Issue 9)

versions of the bot code, and wired it all together correctly. The resulting UI can be seen in Figure 2, which shows the new interface element listing past versions of a user's code with a revert option. While this solution technically worked (the user could revert to past code states), we discovered a serious performance problem: the agent's implementation loaded all stored code versions of all users into memory on server startup (a costly approach). This highlights a scenario where the agent solved the letter of the request but not the spirit of good software engineering – a human developer would likely avoid loading all of the source code into memory. This underscores a fundamental limitation of current LLM agents: they often don't evaluate trade-offs beyond immediate correctness. We marked this solution as not production-ready due to the RAM overhead. It needed human refactoring to be acceptable. Nonetheless, the fact Claude Code managed to implement a multi-step full-stack feature is impressive. For future agents, improving their considerations for performance might be beneficial, although that is a very difficult general problem.

VII. Conclusion

In this paper, we explored the frontier of using state-of-the-art LLMs as autonomous software engineers on real development tasks. Through experiments on ten diverse issues in an open-source project, we found that today's top models can partially fulfill the role of a developer: they wrote code that solved about half of the tested issues without any human assistance. This is a notable achievement and highlights the rapid progress in LLM capabilities for software engineering. However, our study also reveals the clear limitations and challenges that remain:

- Autonomous agents are not yet reliable across the board. They struggled especially with more complex or ambiguous tasks, and often produced suboptimal solutions even when they met the basic requirements.
- Reasoning and chain-of-thought prompting greatly influence success. Utilizing the reasoning ability of Claude Sonnet 3.7 improved outcomes in our trials.
- There is a need for built-in validation and refinement. Incorporating test execution, linting, and iterative self-correction (as Claude Code does) helped catch mistakes. Future agents should leverage all available verification tools (compilation, static analysis, tests) to ensure higher quality outputs.
- Certain errors, like misinterpreting the true intent of a requirement or making inefficient design choices, are currently beyond the examined agents' capacity to avoid. These will likely require changes to the agent's architecture or a human-in-the-loop to guide the agent.

Despite these limitations, the trend is very encouraging. We expect that with each iteration, the gap on what tasks are solvable autonomously will narrow. In practical terms, autonomous LLM agents could already take on some tedious parts of development (like writing boilerplate code, fixing simple bugs, updating configurations), freeing human developers to focus on higher-level design and complex problem-solving.

As future work, our immediate next steps include:

- **Evaluating newer models:** We plan to test open-source QwQ-32B with Aider to see if it can match Claude's performance. If successful, this could open the door to more accessible autonomous coding (not relying on closed APIs).
- **Architect-editor agent design:** We will experiment with an "architect" mode in Aider, where one model (or one prompting style) is used to outline the solution (select files to change, perhaps write pseudo-code or steps), and another model is used as the "coder" to implement those steps.
- **Scaling to more tasks and projects:** Our current test set is small. We want to expand the evaluation to include a wider variety of issues (UI-heavy issues, algorithmic challenges, integration tasks) and on different projects (perhaps some Python backend projects, mobile app issues, etc.). This will paint a fuller picture of where autonomous LLMs excel and where they fail in software engineering.

In conclusion, state-of-the-art LLMs, when coupled with a suitable agent framework, are beginning to demonstrate practical utility in automating segments of software development in a fully unsupervised manner. They function as knowledgeable but flawed junior developers: capable of writing code and solving problems in familiar contexts, yet prone to mistakes that require oversight. By continuing to improve LLM reasoning, integrating robust self-checks, and using clever orchestrations of multiple models, we move closer to a future where AI agents could handle routine programming tasks autonomously. Such a development could significantly accelerate software engineering workflows, allowing human developers to push the boundaries of innovation with the grunt work delegated to our AI collaborators.

REFERENCES

[1] OpenAI, *OpenAI o3-mini*, version 2025-01-31, 2025. [Online]. Available: https://openai.com/index/openai-o3-mini/.

[2] DeepSeek-AI et al., "Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning," 2025. arXiv: 2501.12948 [cs.CL]. [Online]. Available: https://arxiv.org/abs/2501.12948.

[3] GitHub, *GitHub Copilot*, Oct. 2021. [Online]. Available: https://github.com/features/copilot.

[4] N. Nguyen and S. Nadi, "An empirical evaluation of github copilot's code suggestions," in *2022 IEEE/ACM 19th International Conference on Mining Software Repositories (MSR)*, 2022, pp. 1–5. DOI: 10.1145/3524842.3528470.

[5] Anysphere-Inc, *Cursor composer*, 2024. [Online]. Available: https://docs.cursor.com/composer.

[6] S. Yao et al., "React: Synergizing reasoning and acting in language models," 2023. arXiv: 2210.03629 [cs.CL]. [Online]. Available: https://arxiv.org/abs/2210.03629.

[7] Significant-Gravitas, *AutoGPT*. [Online]. Available: https://github.com/Significant-Gravitas/AutoGPT.

[8] J. He, C. Treude, and D. Lo, *Llm-based multi-agent systems for software engineering: Literature review, vision and the road ahead*, 2024. arXiv: 2404.04834 [cs.SE]. [Online]. Available: https://arxiv.org/abs/2404.04834.

[9] H. Yang, S. Yue, and Y. He, *Auto-gpt for online decision making: Benchmarks and additional opinions*, 2023. arXiv: 2306.02224 [cs.AI]. [Online]. Available: https://arxiv.org/abs/2306.02224.

[10] X. Hou et al., "Large language models for software engineering: A systematic literature review," 2024. arXiv: 2308.10620 [cs.SE]. [Online]. Available: https://arxiv.org/abs/2308.10620.

[11] A. F. Khan et al., "Lads: Leveraging llms for ai-driven devops," 2025. arXiv: 2502.20825 [cs.LG]. [Online]. Available: https://arxiv.org/abs/2502.20825.

[12] M. Sapkal et al., "Ai-driven software patch management system," *SSRN Electronic Journal*, Jan. 2025. DOI: 10.2139/ssrn.5086731.

[13] Anthropic, "Claude 3.7 sonnet and claude code," Feb. 24, 2025. [Online]. Available: https://www.anthropic.com/news/claude-3-7-sonnet.

[14] DeepSeek-AI et al., "Deepseek-v3 technical report," 2025. arXiv: 2412.19437 [cs.CL]. [Online]. Available: https://arxiv.org/abs/2412.19437.

[15] Aider-AI, *Aider*, version 0.75.2, 2025. [retrieved: Mar. 2025]. [Online]. Available: https://github.com/Aider-AI/aider.

[16] M. Chen et al., "Evaluating large language models trained on code," 2021. arXiv: 2107.03374 [cs.LG]. [Online]. Available: https://arxiv.org/abs/2107.03374.

[17] Y. Li et al., "Competition-level code generation with alphacode," *Science*, vol. 378, no. 6624, pp. 1092–1097, Dec. 2022, ISSN: 1095-9203. DOI: 10.1126/science.abq1158. [Online]. Available: http://dx.doi.org/10.1126/science.abq1158.

[18] J. Wei et al., *Chain-of-thought prompting elicits reasoning in large language models*, 2023. arXiv: 2201.11903 [cs.CL]. [Online]. Available: https://arxiv.org/abs/2201.11903.

[19] Move-Fast-and-Break-Things, *Aibyss*, 2025. [retrieved: Mar. 2025]. [Online]. Available: https://github.com/move-fast-and-break-things/aibyss.

[20] Y. Mikhalevich, *Aibyss: Issues selected for the "practical applications of state-of-the-art large language models to solve real-world software engineering problems autonomously" research*, 2025. [Online]. Available: https://github.com/move-fast-and-break-things/aibyss/issues?q=is%3Aissue%20label%3Aai-agents-evaluation-2025-03.

[21] Y. Mikhalevich, *Aibyss: Prs created by ai agents as part of the "practical applications of state-of-the-art large language models to solve real-world software engineering problems autonomously" research*, 2025. [Online]. Available: https://github.com/move-fast-and-break-things/aibyss/pulls?q=is%3Apr+label%3Aai-agents-evaluation-2025-03+.