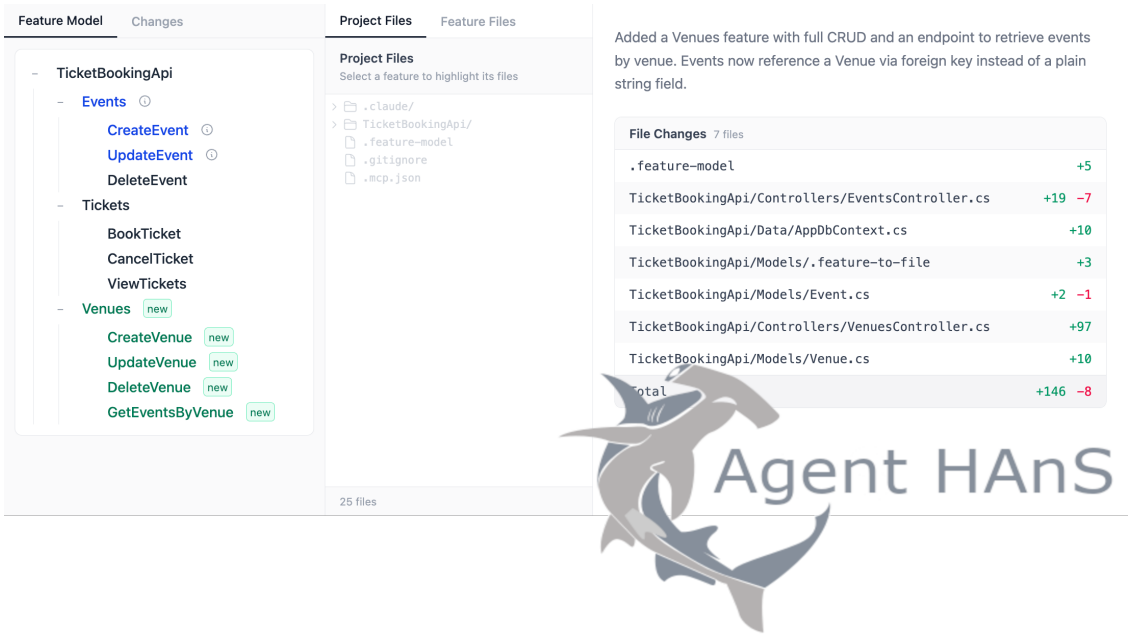


Agent HAnS: Using feature traceability to improve comprehension and trust in LLM-based coding

Master's Thesis in Computer science and engineering

Jacob Bengtsson
Moltas Hultin

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2026

Master's Thesis 2026

Agent HAnS: Using feature traceability to improve comprehension and trust in LLM-based coding

Jacob Bengtsson
Moltas Hultin

CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF
GOTHENBURG

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2026

Agent HAnS: Using feature traceability to improve comprehension and trust in LLM-based coding
Jacob Bengtsson
Moltas Hultin

Supervisor: Thorsten Berger, Department of Computer Science and Engineering
Co-Supervisor: Johan Martinson, Centiro Solutions AB & Ruhr University Bochum
Examiner: Robert Feldt, Department of Computer Science and Engineering

Master's Thesis 2026
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2026

Jacob Bengtsson & Moltas Hultin
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

As LLM-based coding projects grow large comprehension and in turn trust of generated code is reduced. Feature models and embedded feature annotations are established concepts for getting an overview of the features present in a codebase. Both of these however require constant maintenance to reflect their associated codebase. Through a three cycle Design Science Research methodology a MCP-based tool, Agent HAnS, is iteratively implemented and evaluated. Agent HAnS automatically generates and maintains feature models and embedded feature annotations in a LLM-based software development workflow. In addition Agent HAnS presents a graphical dashboard between prompts highlighting changes made to the source code from a feature-oriented perspective. To evaluate the tool, a user study and a correctness evaluation was conducted. The results of the user study show no significant effect on trust and comprehension but that Agent HAnS is usable and does not impact the overall effort needed. The correctness evaluation shows that feature models generated by Agent HAnS varies in quality, sometimes equivalent to a human defined baseline and other times misses critical features. A more extensive study would be better positioned to assess the potential benefits of Agent HAnS.

Keywords: LLM, features, feature models, embedded feature annotations, vibe coding, agentic coding

Acknowledgements

First and foremost, we would like to thank **Johan Martinson** for his wonderful job in guiding us through this master thesis. We really appreciate the late nights and weekends that you have spent on feedback and proofreading despite your other commitments.

We also want to thank **Thorsten Berger** for his valuable feedback during the project and helping us laying a strong foundation for the thesis.

Furthermore, our thanks goes out to **Kevin Hermann** for offering his time in shaping parts of our thesis and his sharp suggestions.

Thanks to **Centiro Solutions AB** and **Team Nova** for hosting us and to all developers at Centiro who participated in the interviews and user studies.

A final thanks goes to **Claude**, without whom we wouldn't have been able to complete the thesis.

Jacob Bengtsson & Moltas Hultin, Gothenburg, May 2026

Use of AI in this thesis

This thesis' topic is about LLM-based coding, an inherently AI-based workflow. To familiarize ourselves with LLM-based coding much of the code generated during this thesis has been generated through the means of large language models. None of the text in the report has been generated by AI tools, but AI tools have been used in the writing process to answer questions.

Links

Tool source code:
`https://github.com/isselab/Agent-HAnS`

Dataset for correctness evaluation:
`https://github.com/isselab/vibe-coding-dataset`

Production Babies

Ted Svante Martinson

Contents

List of Figures

List of Tables

1

Introduction

Since the release of ChatGPT in 2022 use of LLMs in daily life has become more and more common. In software engineering, AI tools quickly found their place enabling code completions and even writing entire functions and applications. In recent years the AI coding landscape has evolved quickly with tools like Cursor [7], Github Copilot [18] and Claude Code [5] becoming main stream. These tools go beyond just generating the odd function, instead using powerful agents to create entire software systems from scratch based on natural language prompts. This paradigm, colloquially referred to as vibe coding [23], describes a development workflow in which the developer delegates substantial portions of the implementation to an AI system, guiding it through natural language prompts rather than writing code directly.

While this approach enables rapid code synthesis it also introduces a new challenge where the developer can over time lose understanding of the behavior and structure of the codebase being generated [48, 41]. This in turn can lead to the developer's mental model and the true functionality of the software to drift apart. This loss of comprehension is not just an inconvenience, it has consequences on software quality and maintainability. Reasoning about the software becomes much more difficult without a proper understanding of the underlying code. This hinders developers verifying the correctness, assessing the impact of future changes and judging the current capabilities of the software.

To mitigate this issue a number of strategies have been proposed. Mitchell et al. [32] describes how using formal methods can create structure by verifying each step in a vibe coding process. Maes [28] on the other hand mention using classic techniques like code reviews and tests. These methods provide structure and correctness to the code generation process, but does not invite the developer into understanding. A systematic tool that provides developers with ongoing support for comprehension and trust, going beyond gate checks for a coding agent to pass, could prove effective.

A well known concept for developer comprehension in the domain of feature-oriented software engineering are *feature models* [37]. Feature models use a simple tree structure to show the structure and relationships between the different capabilities, or features, present in a software system. They act in a layer of abstraction above the granularity of individual functions to provide an overall understanding of a codebase. However, feature models also have their challenges. Constant manual effort is

required to maintain them to ensure their structure reflects the true structure of the codebase they intend to represent [10]. Just like with other disconnected types of documentation, becoming disregarded and getting out of sync is a problem requiring constant attention.

Despite this potential of using feature models to provide comprehension for LLM-based coding and for generative AI to alleviate the burden of having to maintain feature models manually, no solutions utilizing both LLMs to automate code generation and maintenance of feature models exist. This leaves a gap which this thesis aims to fill. We present Agent HAnS, an agentic successor to HAnS (Helping Annotate Software) [30], an AI-coding support tool that maintains a feature model and displays it to the developer using a web-based dashboard. By seamlessly integrating feature traceability into LLM-based coding workflows we provide a continuous high level awareness of the software system without significant additional effort required from the developer.

1.1 Purpose of the Study

The purpose of this study is to bridge the comprehension gap in LLM-based coding by engineering and evaluating a tool that automatically generates and maintains feature models and embedded feature annotations throughout the development process. By introducing feature traceability into the development workflow, the tool could mitigate some of the challenges that developers otherwise face when using an LLM-based workflow. Specifically, this study seeks to determine if an automated, low-effort approach can successfully generate accurate feature models that enhance developer trust and understanding without disrupting the rapid coding flow.

1.2 Contribution

This thesis makes three contributions to the field of LLM-based coding.

First, we present Agent HAnS[1], a tool that automatically generates and maintains feature models and embedded feature annotations during LLM-based coding. The tool can be integrated into existing workflows without requiring developers to change how they use their existing tools. It also provides a graphical dashboard giving developers continuous visibility of the changes made to their code base over time.

Second, we provide empirical evidence on how automated feature traceability affect developer trust and comprehension during LLM-based coding, through a user study.

Lastly, we evaluate the quality of LLM-generated feature models through a correctness evaluation, and contribute a dataset[2] of six LLM-generated projects and the conversations that generated them.

[1] https://github.com/isselab/Agent-HAnS
[2] https://github.com/isselab/vibe-coding-dataset

1.3 Research Questions and Method

This thesis is based on a three-cycle Design Science Research project. The DSR process consists of three main parts: (i) problem understanding, (ii) proposing a solution, an artifact, (iii) evaluation of the solution. The evaluation involves iterative prototyping, a user study with developers at Centiro Solutions AB to assess trust and comprehension, and a correctness evaluation to assess the quality of generated the feature models.

RQ 1: How can a tool be designed that leverages feature models and embedded feature annotations in order to make LLM-based coding techniques more comprehensible and trustworthy?
The tool will be designed through three cycles of Design Science Research.

RQ 2: How effective is the tool in supporting developers during LLM-based coding?
RQ2 will be answered through its sub-questions.

- **RQ 2.1:** How does the tool affect developers' understanding and trust of LLM-generated code?
  Understanding and trust will be assessed via survey and code comprehension questions.

- **RQ 2.2:** What is the quality of LLM-generated feature models and embedded feature annotations?
  Quality will be assessed in terms of accuracy and completeness compared to a defined baseline.

- **RQ 2.3:** How do the users of the tool perceive its usability?
  Usability will be evaluated using the System Usability Scale (SUS) and qualitative feedback.

1.4 Delimitations and Limitations

Delimitations. A core delimitation, or premise, for this thesis is that feature models will be used to improve trust and comprehension in LLM-based coding. Other approaches to the problem are not considered.

Agent HAnS is designed to be used when building new projects, or when working on projects where feature models and embedded feature annotations are already a part of the code base. It has not been designed to scan and annotate existing code bases retroactively.

Limitations. The primary limitation of this thesis is time. The thesis was conducted over a period of 20 weeks, which put a time limit on both the implementation and evaluation.

The thesis is further limited by the models and providers available to us. The specific

models used during the study can be seen in chapter 3.

2

Background and Related Work

This chapter establishes the theoretical foundation for feature-oriented software engineering and LLM-based coding, with an integrated review of related work to highlight current gaps and research. The first two sections cover relevant concepts relating to feature oriented software engineering and LLM-based coding. The chapter concludes with a review of the literature that describes the gap which this thesis aims to fill.

2.1 Feature-Oriented Software Engineering

One of the premises for this thesis is that it will make use of feature models. Improving LLM-based coding can be approached from many different directions, but here this specific direction was chosen from the start. Feature-oriented software engineering stems from the notion that evolution of software becomes easier to understand when thinking at the level of features [37]. In the landscape of feature-oriented software engineering multiple concepts and frameworks exist to make working with and reasoning about features easier. Some of the relevant concepts to this thesis are presented below.

2.1.1 Feature Models

Feature models are a tree-based hierarchical way to describe the relations between different features. A feature is an abstract way to describe a specific functionality or characteristic of a software product, where the concrete side of it is the implementation in code [9, 15, 26, 25]. Dividing software into features enables the general concept of software variants where a singular code base can offer different functionalities based on what features are activated or not. A typical example is that of a printer company, where two different printers mostly have the same functionality: printing paper. Still, the more expensive model might have access to features like scanning or color printing. Instead of maintaining two separate code bases for these two printer models, variants of a singular code base reduces code duplication.

This notion of dividing software into features to enable effective use of software variants was popularized by Clements and Northrop [13] under the term *Software Product Lines*. While both currently and historically being a major reason for di-

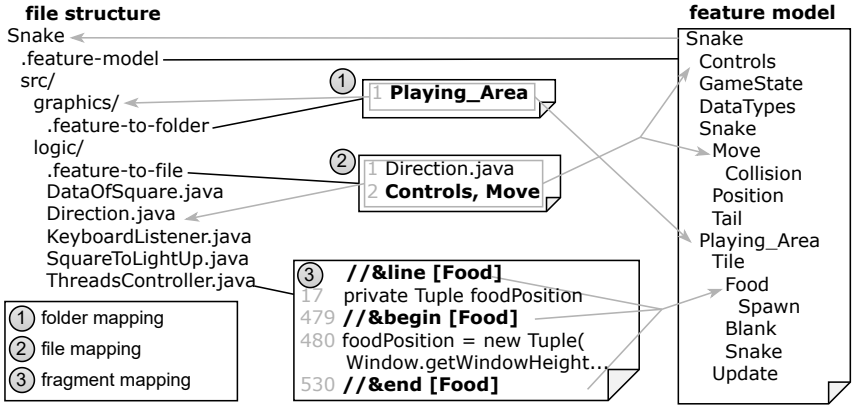


Figure 2.1: Example of feature annotations using embedded feature annotations. Figure from Martinson et. al. [30]

viding software into features, SPLs focus on product generation. More recent work has shifted towards using feature models for maintaining traceability in evolving codebases which is described in the following sections.

2.1.2 Embedded Feature Annotations

The feature model describes what features exist in the code base, but not where they are located in the code. Feature location is one of the most common and time-consuming tasks for developers [42, 38, 36, 10, 26, 47]. Individual features are often scattered throughout the code base and forgotten as developers work on different tasks or leave and get replaced. Automated feature location techniques have shown to be time-consuming and produce many false positives [36, 47, 39]. Therefore, previous research suggests that developers should trace a feature's location while the locations are still fresh in their minds [22, 21, 30]. One way to create these traces is by using embedded feature annotations [44]. This is explained and explored as a way to make features explicit in code in many studies [27, 22, 30, 29, 44, 21, 2].

Figure 2.1 shows an example of embedded feature annotations. There are three main ways that we can trace features: folder mapping, file mapping and fragment mapping. File and folder mappings use separate files to track features whereas fragment mappings use block or inline comments.

2.1.3 Application

In this thesis the main benefit of maintaining a feature model and embedded feature annotations is keeping a record of what features are implemented in a code base as a documentation tool [10]. This helps developers, especially developers new to a project, see what features exist, how they relate to one another and where they are located in the codebase. This benefit also applies to non-technical people who interact with the project as they can get a high-level overview of the implemented features and their position in the project. Despite these benefits, organizations often find adoption difficult, due to the manual effort required. Martinson et al. [31]

proposes a structured process to work with feature models and embedded feature annotations. This process is still manual and leaves a gap for using LLMs to make introducing feature models automatized and easier, as a motivation for this thesis.

2.2 LLM-based Coding and Vibe Coding

In comparison to traditional coding where the developer primarily works with a programming language to create software, LLM-based coding is performed through higher level abstractions, mainly natural language [43]. Through dialogue with a large language model, code is generated and the generated result is then evaluated by the developer. Through further natural language dialogue code is generated and modified without the developer having to directly write any of the code themselves.

During the process of writing the thesis, we often used the word vibe coding, but found that the developers at Centiro had different ideas of what vibe coding is. Some considered all AI-assisted coding to be vibe coding, like code completions in IDEs. Others followed a stricter definition where the majority of developer interaction stays within the prompt window and looking less at the code itself. This led us to realize we needed to have expressed opinion of what type of LLM-based coding we're trying to target in this thesis.

The origin of the term *vibe coding* is credited to Andrej Karpathy [23], where the developer embraces the so called *vibe*, forgetting that the code exists at all. Reading the literature seems to indicate that vibe coding has taken a broader definition, as being LLM-first coding, where the developer mainly interacts with the prompt window, while still looking at the code at times. This is in comparison to Karpathy's definition which could be said to be LLM-only coding.

Finally, the concept of agentic coding exists in literature, and a description and differentiation of it is given by [43]. Agentic coding became a new LLM-based coding pattern where the developer can define different agents for different tasks in development. This can be seen as an extension to vibe coding, where the prompt window was the only means of instructing LLMs what code to generate in the beginning, but now there are additional ways to constrain and direct the LLM. Many of the major LLM coding tools, like Anthropic's *Claude Code* [5], Anomaly's *OpenCode* [3] or GitHub's *Copilot CLI* [18] allow the developer to use and define a multitude of different agents, as well as many other features described below.

In this thesis we're looking at the broader definition of vibe coding where agentic coding is included and the developer is allowed to look and interact with the code directly, while the majority of code is generated through a prompt window by an LLM.

2.2.1 LLM Extension Components

There are many ways to change how an agent acts by giving it additional instructions, beyond just sending prompts. Delivering these instructions can be done through different means depending on the type of instruction and interaction. These means of instructing agents are essential to get them to understand how feature models and embedded feature annotations work and how they should be created.

Agents. Agents are at the core of LLM-based coding. They are responsible for taking a prompt and turning it to code written on disk and executing it [43]. To generate said code and reason about its decisions agents use LLMs as a reasoning engine. An LLM on its own is just a mathematical prediction function with no way to interact with a user's computer on its own.

More specialized agents can be defined by the developer targeting a variety of tasks with customizable levels of permissions. The agents are defined mainly through natural language describing how they should act and what their purpose is. A more specific agent could be an agent which writes unit tests. It's definition can contain instructions how test should be generated and what style guidelines to follow. It is also possible to have agents which are specialized in non-software engineering tasks. Agents can be instructed to think about other aspects of a software project, like environment, economics or legal topics for instance.

Agentic tools, like OpenCode, also have the ability to during a single prompt delegate tasks to subagents. The benefits of delegating are that it is possible to perform parallel execution of tasks and to reduce the load on the main agent's context window.

Skills. Skills are a complement to agents [6, 4]. Skills are created through natural language and are intended to provide instructions for a specific task available to any agent. Agents execute in their own context and have their own permissions, whereas skills are just extra instructions that can be accessed by an agent. Skills can not be executed on their own, they need to be accessed through an agent. Each skill has a short description which helps the language model quickly identify what skills are available without needing to put the full instructions contained in the skill into context. It can be thought of like a bookcase, where agent can choose to read a book of their choosing based on the titles, without having to commit the entire bookcase to memory all the time.

The benefit of skills is that they allow a language model to selectively expand its available instruction set during runtime, without having to put all skills into the model context by default, as the context space is a limited resource.

Deciding whether to define a specific instruction as an agent or a skill is not always obvious. In thesis the different iterations of the artifact moved from using subagents to using skills in some cases.

Global instruction file. A global instruction file (often called AGENTS.md or

CLAUDE.md) is a file that exists either as a file in the root of a project, or as a truly global file in the configuration of a tool. It instructs all agents how to act. In collaborative projects they can be used to ensure all developers use the same style guides when generating code using LLMs. The advantage with a global instruction file is that it does not require any active action from the user to come into play.

Model Context Protocol. The model context protocol is an open-source standard for connecting AI-models to external systems [35]. The standard consists of a common protocol that MCP servers and clients can utilize.

MCP clients are built into the application connecting to a MCP server [33]. The applications can be chat applications, IDEs or other LLM based applications. The client communicates with the server and provides the server with core features the server can utilize when running its tools.

MCP servers are similar to skills in the sense that they only expose a short instruction but offer extra functionality on request [34]. MCP servers are more powerful than a skill as they involve running deterministic code. This can enable the LLM to interact with systems that ordinarily the LLM doesn't understand or have access to. A typical example for a MCP server is one that interacts with a database, making it possible for the LLM to directly interface with the database creating new entries and modifying tables in a more deterministic fashion. To do this the different functions, or tools, of the MCP server need to be described through natural language over the protocol.

2.3 The Gap in LLM-based Feature Traceability

While LLMs have shown promise in reengineering software product lines [1] and improving feature extraction [45], these approaches focus on high-level SPL generation or require significant human oversight. No existing work addresses the real-time, agentic generation and maintenance of fine-grained feature models and embedded annotations during active development. Furthermore, the specific human-AI collaboration framework required for this domain, distinct from general requirements engineering [40], remains unexplored. This thesis fills this gap by focusing on real world developer workflows using the AI coding tools of today and how the notion of features can be used to make that process more comprehensible and trustworthy.

3

Method

This thesis followed the Design Science Research (DSR) methodology as described by Kuechler and Vaishnavi [46]. The idea behind design science research is to address identified problems through the iterative design and development of artifacts. In addition, the study applied the guidelines for DSR for master thesis projects in industry as described by Knauss [24]. This chapter describes the problem understanding process and the guiding principles formed from it. Together they form the basis for the implementation of the artifact (Agent HAnS). Finally the evaluation methods, a user study and a correctness evaluation, are described.

3.1 Design Science Research

We split the work into three separate cycles, each of which included three phases: problem understanding, solution and implementation, and evaluation. For each cycle the focus shifted between the different phases. This is visualized in table 3.1 along with the work conducted.

3.1.1 Problem Understanding

The majority of the work related to problem understanding was performed during the first cycle. This was done by gathering information from literature and by conducting interviews with industry practitioners. We also conducted a small benchmark. The primary objective was to identify specific challenges in LLM-based coding, which together with stakeholder input at Centiro formed the basis for a set of guiding principles for the artifact.

Interviews. The interviews were conducted with developers at Centiro in order to get a better understanding of how they used LLM-based code generation in their everyday work. We recruited seven professional developers by first asking a few developers in the team we were placed in at Centiro and then letting each interviewee recommend people who they thought we should interview. The developers had varying amounts of experience with AI-coding tools, some used AI-tools daily, whereas others had only encountered tools but weren't frequent users. In regards to professional experience four of them had worked for around five years, whereas the other three had worked for more than twenty. These insights directly informed

Table 3.1: Visualization of the three cycles by research questions. The shaded cells show where the emphasis lies in each cycle. Adapted from Knauss [24]

Cycle	Problem	Solution	Evaluation
1	Interviews and analysis of LLM-based coding and identification of challenges.	Brainstorming and prototyping of the initial artifact. Validate base idea with stakeholders.	Small demo round with stakeholders on prototype artifact
2	Further discussions based on result from previous cycle	Workshops based on feedback from stakeholder demos. Validate full feature set with stakeholders.	Demos with stakeholders on fully featured artifact.
3	Minimal discussions based on result from previous cycle	Implementation based on feedback from previous stakeholder demos	User study and correctness evaluation on revised fully featured artifact

guiding principles #7 and #8 regarding comprehension and trust. The questions for the interviews can be found in appendix A.

Benchmark. The benchmark was conducted on a selection of available LLMs to see how they performed at feature model related tasks. This benchmark was done as a baseline reading in order to give a picture of how different models performed and guide which model was to be used during future trials.

We decided to test a model from each of the main players in the AI space: OpenAI's *GPT5.2*, Google's *Gemini 2.5 Pro* and Anthropic's *Claude Sonnet 4.5*. The models were tasked with three different tasks. In the Snake project[1], a part of the EASElab datasets of pre-annotated code bases [16], they were asked to add, remove and rename a feature as three separate tasks. The changes made by the LLM were then compared to a baseline created by us in advance, checking if relevant code was added and annotated correctly, and if the feature model was updated correctly. The tests were repeated 10 times per model to combat the inherent non-determinism of LLMs. In short all models completed all tasks with a complete success rate, meaning no model performed better than any other and as such the selection of models used came down to other factors. We mainly used Anthropic's *Claude* models, as they were offered by Centiro. For more detailed results and discussion of the benchmark see appendix B.

Challenges. Based on the literature, the benchmark, and the interviews, challenges with LLM-based coding were identified. The challenges were split into three categories: (i) limitations from providers, (ii) how models act, and (iii) user familiarity, understanding, and trust. All identified challenges can be found in appendix C.

[1] https://bitbucket.org/easelab/datasetsnake/src/master/

Table 3.2: Guiding principles for the design of the tool

#	**Principle**	Origin
1	The tool must display the feature model visually	Stakeholder Discussions
2	The tool must show the mapping between features and their corresponding files	Stakeholder Discussions
3	The tool must be able to generate and maintain a feature model for a codebase	Stakeholder Discussions
4	The tool must be able to generate embedded feature annotations in source code	Stakeholder Discussions
5	The tool must be compatible with different AI coding tools	Stakeholder Discussions
6	The tool must integrate well into existing development workflows without meaningfully disrupting the typical vibe coding experience	Stakeholder Discussions
7	The tool must address the challenge of developer comprehension of LLM-generated code	Identified Challenge
8	The tool must address the challenge of developer trust in LLM-generated code	Identified Challenge

Of these, two were selected as the most relevant: understanding generated code and lack of trust.

Guiding Principles. Based on the relevant identified challenges and discussions with stakeholders at Centiro, a set of guiding principles were established. These guiding principles act as both requirements and guidelines for the design of the tool, and are presented in table 3.2.

3.1.2 Solution and Implementation

For each cycle, a solution was implemented based on the insights gathered in the preceding problem understanding phase.

In the first cycle, several different architectures were ideated, explored and discussed based on the guiding principles. Architecture in this case refers to what different LLM components and tools to make use of, such as agents, skills, MCP servers or plugins. For example, in the final version of Agent HAnS, the architecture consists of a global instruction file, two skills and an MCP server. Once an architecture was decided upon the solution was implemented in code. For this thesis a Node.js TypeScript-based implementation was used.

For the second cycle, stakeholder feedback and experiences when testing the prototype resulted in an expanded set of features for the tool: feature to file associations and using Git's *diff* function to show how many lines affected a specific feature were added. The architecture from the first cycle still had the capability of handling the new features, with a slight modification to some of the steps in the LLM workflow, as we moved from using a subagent and a skill to instead only using skills.

In the final cycle, the artifact underwent only minor refinements. These changes included small adjustments to the workflow and resolving minor issues identified during testing, such as the global instruction file including a check that a Git repository is initialized. The final version of the implementation is described in full in chapter 4.

3.1.3 Evaluation

The main evaluation was conducted in cycle three after the tool's iterative development was considered finished. This evaluation was divided into two parts: a user study and a correctness evaluation. The user study focused on understanding how developers interacted with the tool, including their perceptions of usability and trust. The correctness evaluation focused on assessing the quality of the generated feature models.

Together, these evaluation approaches provided both qualitative and quantitative insights into the effectiveness of the proposed tool. While the user study captured the developers' experiences and perceptions, the correctness evaluation examined the technical quality of the produced artifacts. The two evaluations are covered more in depth in the following two sections.

For cycle one and two, the evaluation phase focused on formative evaluation consisting of internal testing and stakeholder demos. Through internal testing, we validated that the tool behaved as intended, for example verifying that the summary displayed in the graphical dashboard was reasonable and that the feature model was updated as expected. The stakeholder demos allowed us to present the current state of the tool and collect feedback on both the design and functionality, which was then used to inform the next cycle. The feedback consisted mainly of wishes to change the layout of the graphical dashboard and additional features. This included both features already planned, like the feature to file associations, but also new features like making use of Git's *diff* function.

3.2 User Study

This section describes evaluation designed to address RQ2. We conducted a within-subjects crossover user study comparing LLM-based coding with and without access to Agent HAnS. The study assessed three dimensions: code comprehension (RQ2.1), feature model quality from a user perspective (RQ2.2), and usability (RQ2.3). The following sections describe participant recruitment, study design, task design, mea-

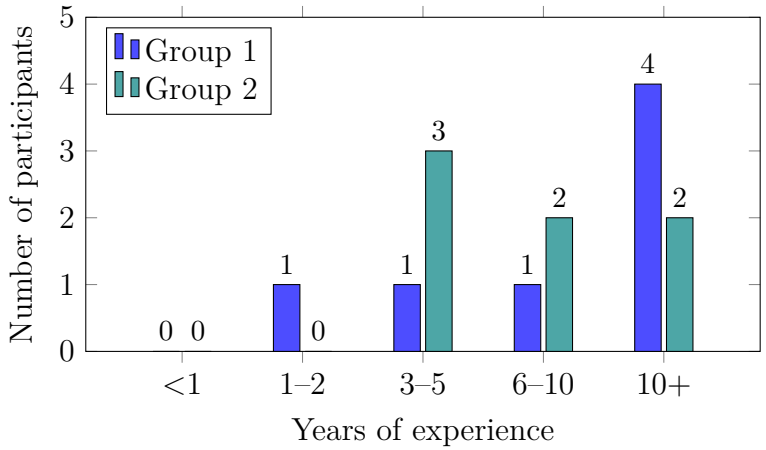


Figure 3.1: Distribution of years of professional development experience per group.

Table 3.3: Background familiarity with different concepts by group (1-5 scale).

	Group 1		Group 2	
Familiarity with:	**Avg**	**SD**	**Avg**	**SD**
LLM prompting	3.9	1.5	4.4	1.1
The notion of features	2.9	1.2	3.9	1.2
Feature models	2.0	1.4	3.1	1.4
Embedded feature annotations	2.9	1.4	2.9	1.2

surement instruments, and the analysis.

3.2.1 Participants

We recruited participants from Centiro Solutions AB. We invited approximately 200 software developers to participate. The study was designed to be completed from anywhere in the world, providing all instructions through an online form. In total, 14 developers participated in the user study. We divided the participants evenly into two groups of seven. Figure 3.1 shows the participants' professional experience by group and table 3.3 shows their familiarity with different concepts related to the user study.

3.2.2 Study Design

We designed the user study to cover all three subquestions of RQ2. Each participant completed two tasks, where one acted as the control and the other as the treatment. In the treatment task the participants had access to Agent HAnS, and in the control task they did not. We used a within-subjects crossover design so that every participant experienced both conditions, meaning each participant effectively act as their own control and individual differences between participants are less likely to skew the results. We therefore split the participants into two even groups, where group 1 had access to the tool during task 1 and group 2 during task 2. This is visualized in table 3.4.

Table 3.4: Visualisation of crossover design.

	Group 1	Group 2
Task 1	With tool	Without tool
Task 2	Without tool	With tool

The study proceeded in six steps:

1. **Consent and setup**: Participants provided consent to the data collection and were instructed on how to download the two repositories needed.

2. **Task 1**: Participants completed four development steps for task 1. Group 1 had access to Agent HAnS. After each step, participants committed their work and answered a code comprehension question about the changes just made.

3. **Post-Task 1**: Participants answered a NASA-TLX questionnaire and a set of domain-specific questions. Group 1 additionally completed a SUS survey to evaluate the usability of Agent HAnS.

4. **Task 2**: After a 5 minute break, participants completed four development steps for task 2. Group 2 had access to Agent HAnS. The same comprehension question structure as task 1 was followed.

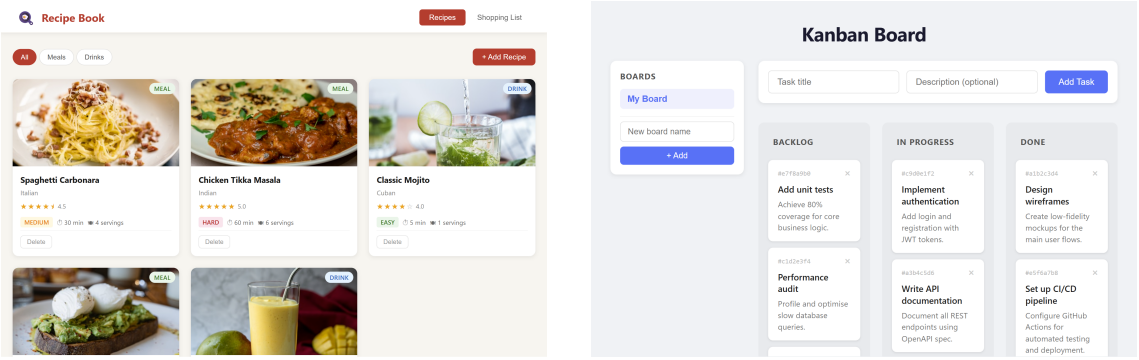
5. **Post-Task 2**: Participants again answered a NASA-TLX questionnaire and a set of domain-specific questions. Group 2 additionally completed the SUS survey.

6. **Final questions**: After both tasks, participants answered questions about the quality and relevance of the feature models and embedded feature annotations generated by Agent HAnS. They also answered background questions covering years of professional experience and prior familiarity with LLM prompting, feature models, and embedded feature annotations.

3.2.3 Tasks and Materials

We created two projects to act as a starting point for the tasks. Task 1 featured a recipe book and task 2 featured a kanban board, see fig. 3.2. Each project started at approximately 600 lines of code, with 10 and 15 features defined in the feature model respectively. We chose to use React as the technology to ensure that no additional tooling was required to run them. Since having access to a JavaScript runtime (e.g. Node.js) is a requirement for using Claude Code, we could ensure that the tasks would work for all participants.

Each task consisted of four development steps. Each step provided a written description of the functionality to be added or changed. Participants were asked to use Claude Code to implement the changes without writing any code manually. The steps were all meant to simulate a realistic development cycle. After completing each



(a) Task 1: Recipe app **(b)** Task 2: Kanban board

Figure 3.2: Screenshots of the initial state of the two tasks.

step, participants were asked to commit their work before moving on. This commit structure allowed the generated code to be inspected per step during analysis.

Each project existed in two variants, one with the tool enabled and one without. This allowed participants to simply download the correct repositories for their group assignment without worrying about tool installation.

3.2.4 Metrics and Measures

Code comprehension. After each step in a task, participants were asked a code comprehension question related to the changes made in that step. The questions were answered in free text, allowing participants to express their understanding in their own words. Each answer was then compared against the generated code for the corresponding step to determine whether it was correct or incorrect. This allowed us to assess the degree to which participants understood the changes made by the LLM.

NASA-TLX. After each task, participants were asked a set of NASA-TLX questions. NASA-TLX (Task Load Index) is a subjective workload assessment tool [20]. The questionnaire consists of six questions on a scale from 0 to 100, where 100 indicates a high workload. In the original NASA example there is also a weighting step where the different subscales are weighed against one another on their importance. We chose to go with unweighted, or raw, NASA-TLX, since the weighting step has been shown to not affect validity or sensitivity [19]. This unweighted total score is obtained by averaging the six subscales. The NASA-TLX score is used to answer RQ 2.3, how the tool affects the overall workflow in terms of overhead and usability.

Domain-specific questions. After each task, the participants were asked a set of domain-specific questions. The participants were given six statements each answered with a Likert-scale response from 1 to 5. The first three statements intend to give an indication of how Agent HAnS affects user comprehension and understanding of the generated code. Statements 4 and 5 intend to look at user trust in the generated code. The final statement addresses both comprehension and trust. Note that the

fourth statement is negatively phrased, implying that the aim is low agreement.

- I have a good understanding of what the generated code looks like and how it is structured.
- I could have explained the changed code to another developer.
- I knew where I would look if I wanted to verify the changes that were made.
- I felt uncertain about what was happening in the codebase at some point during the task.
- I trust that all parts of the codebase relevant to my request were addressed.
- I would feel confident handing this codebase off to another developer.

System Usability Scale (SUS). The System Usability Scale consists of ten statements, with Likert-scale responses from 1-5 [12]. The statements alternate describing positive and negative usability aspects. SUS scores were calculated by subtracting 1 from the scores of the positive items and subtracting the scores of negative items from 5. The adjusted item scores were summed and multiplied by 2.5, resulting in a total SUS score ranging from 0 to 100. A SUS score of 70 or above was found to align with the adjective *good* according to [8] and is the threshold which will be used. The SUS score will be used to evaluate the usability of Agent HAnS and as a part of the results of RQ 2.3.

Feature Traceability Questions. Finally, the participants were asked two questions about feature traceability. While the correctness evaluation in the following sections is the primary way RQ 2.2 is answered, these questions provide an additional qualitative perspective. The questions were phrased as statements with Likert-scale responses from 1-5.

- The features shown in the feature model felt relevant and of quality.
- There was code that belonged to a feature that was not annotated.

3.2.5 Analysis

The analysis was structured to address each subquestion of RQ2. Code comprehension questions and domain-specific questions address RQ 2.1, NASA-TLX and SUS address RQ 2.3, and the feature traceability questions address RQ 2.2.

Code Comprehension. All text responses to the code comprehension questions were evaluated by comparing the answer with the corresponding step's commit. Each answer was labeled by us as either correct or incorrect. After labeling, the correct answers were summed up per condition and task.

NASA-TLX and Domain-specific questions. Since each participant completed both conditions, we used a paired t-test [14] to compare within-subject dif-

ferences in NASA-TLX scores and domain-specific Likert responses. Statistical significance was assessed using a significance level of $\alpha = 0.05$, meaning results with $p < 0.05$ were considered statistically significant. Effect sizes were reported as Cohen's d [14], where values around 0.2, 0.5, and 0.8 indicate small, medium, and large effects respectively. To check for order effects introduced by the crossover-design, we compared NASA-TLX and domain-question scores between tasks regardless of condition.

System Usability Scale. SUS scores were calculated as described in section 3.2.4 and interpreted using the adjective rating [8], with a threshold of 70 corresponding to *good* usability.

Feature Traceability Questions. The result of the feature traceability questions were analyzed descriptively, with mean and standard deviation being calculated.

Qualitative Feedback. The end of the survey has four open-ended questions. The responses were analyzed and compiled in order to inform the discussion.

3.3 Correctness Evaluation

To evaluate the ability of Agent HAnS to create correct and relevant feature models a second benchmark was conducted. The smaller benchmark during problem understanding served mainly as a way to show if certain LLMs performed better at tasks related to feature models. The tasks performed in the benchmark were too simple to draw any insights to how well LLMs handle creating feature models in more realistic and complex scenarios.

To do this we broadened the scope by letting agents define projects from scratch, compared to the small benchmark where only minor changes were made to an existing project. The projects themselves were also larger in scope and more complex requests were given to the agents. Due to the more ambitious scope the benchmarking process could not be automated. The feature modeling process becomes too variable for larger projects, meaning we could not define a ground truth that we could reasonably expect the agents to match perfectly and programmatically verify against. Additionally, perfectly matching the ground truth isn't necessarily the goal, rather, the goal is relevancy: *Is the generated feature model as relevant or useful as the ground truth?* To achieve this we had to define the ground truth feature models manually and then using a set of grading criteria compare generated feature models against the ground truth and give them scores accordingly.

3.3.1 Tasks and Materials

For this benchmark six different projects[2] were defined and then generated through a series of prompts. After a number of prompts a feature model was defined manually using FM-PRO [31]. We identified features using the bottom-up activity and

[2] `https://github.com/isselab/vibe-coding-dataset`

created the model by first defining a coarse hierarchy and adding features to the hierarchy [11]. The development then continued with more prompts until the next break point where another feature model was recorded using the same procedure. After the project was finished a third and final feature model was defined. Together this resulted in three different feature models for each project at three different *evaluation points*. These manually defined feature models act as a baseline to compare the other two models against, a ground truth. The complete conversation from generating each project was then exported.

Next, reusing the same prompt sequence all projects were generated again but with Agent HAnS enabled. At each evaluation point the current feature model generated by Agent HAnS was recorded. Finally, using the below prompt, a third set of feature models were extracted by from the previously exported conversation by an LLM, specifically *Claude Sonnet 4.6*. The prompt is intended to be minimal as to not affect how the agent acts when extracting the feature models. We wanted an as pure representation as possible to find what a feature model looks like without the tailored instructions included in Agent HAnS. Still, some additional instructions had to be included in the prompt. Without the instructions about constraints feature models more aimed towards SPLs were generated, which wasn't what we were after.

> *Inside this folder there is a text file, which is an exported conversation from Claude Code. It consists of three main feature prompts, and a few other prompts to ensure functionality and fix bugs. I want you to create a feature model for each of these three stages. I'm not considering constraints, I want a tree-style structure with parent features and their children, to show what features are available at each stage. Each stage should build on the previous stage to give a complete snapshot of what the software structure looked like at that moment.*
>
> Extraction prompt

Together these three sets are the basis for the correctness evaluation, where the first set acts a baseline to compare the other two sets against. Figure 3.4 shows an example of what the three sets can look like. The complete workflow to define the three sets of feature models for each project is visualized in fig. 3.3. Tables 3.5 and 3.6 show an overview of the projects.

3.3.2 Analysis

To help compare the feature models more systematically a set of grading criteria was defined. The scale is numbered, but should not be considered linear. The following grading criteria were used to evaluate the correctness of the feature models:

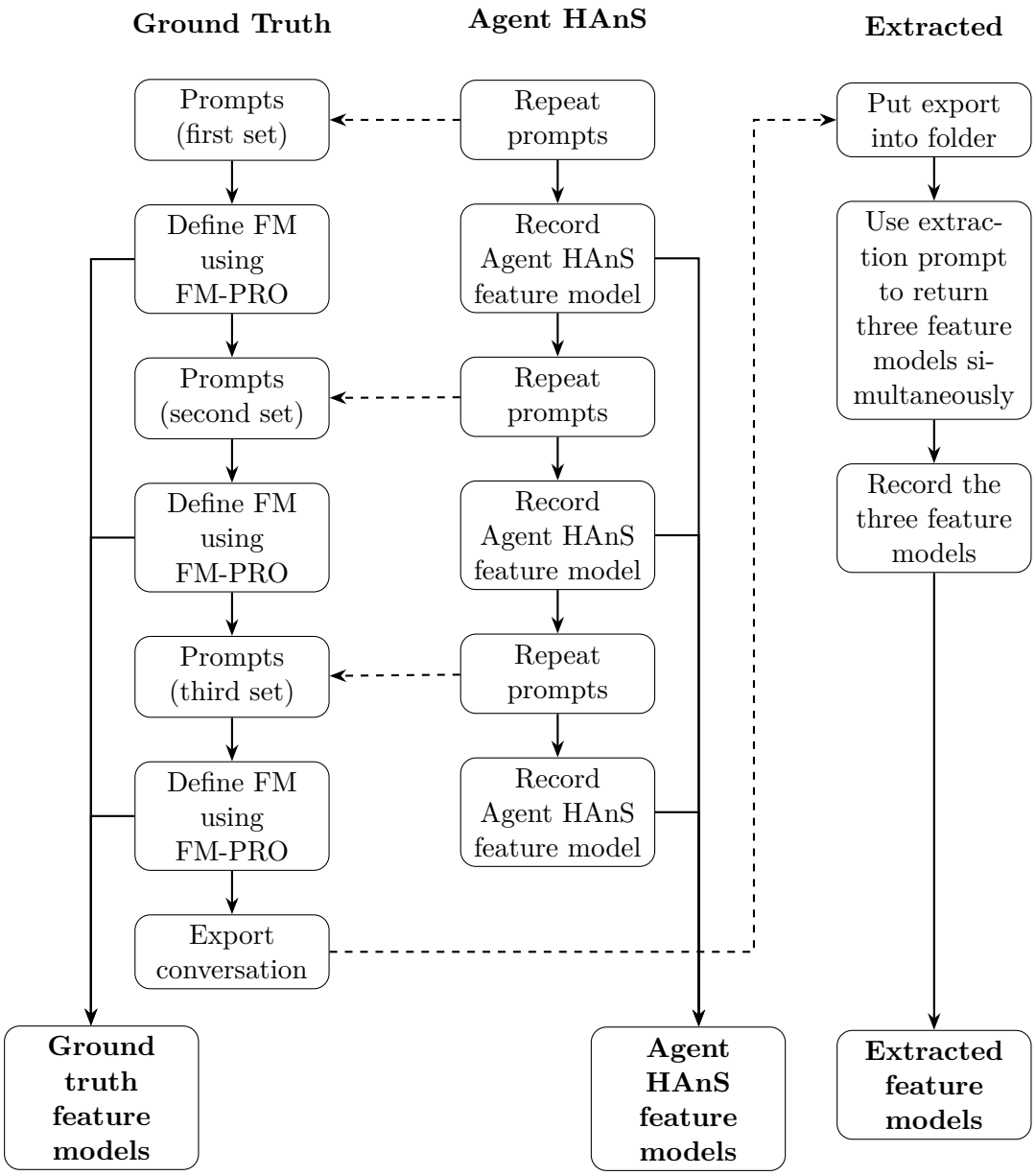


Figure 3.3: Flow for the correctness evaluation

Table 3.5: Overview of projects used in the correctness evaluation (1/2).

Battleship		
Ev.	**Feature model change**	**Type**
1	Grid	Add
	Ships	Add
	Ship placement	Add
2	Multiplayer	Add
	Gameplay — Single room	Add
	Ship placement → Click, Drag and drop	Split
	Realistic ship designs	Add
3	Gameplay — Single room → Gameplay — Multiple rooms	Rename
	Placement mode selection — Click	Remove
	Shell firing animation	Add
	Ship sunk animation	Add
Timer App		
Ev.	**Feature model change**	**Type**
1	TimerList	Add
	Timer	Add
	Timer — StartStop	Add
	Timer — Reset	Add
2	TimerList → List	Rename
	Stopwatch	Add
	Stopwatch — StartStop	Add
	Stopwatch — Reset	Add
3	Stopwatch — Lap	Add
	Keybinds	Add
Noticeboard		
Ev.	**Feature model change**	**Type**
1	Text card	Add
	Image card	Add
	Drag to reposition	Add
2	Multiplayer	Add
	Admin role	Add
	Visitor role	Add
	Emoji reactions	Add
	Text card, Image card → Unified card	Merge
3	Single noticeboard → Multiple noticeboards	Rename
	Board themes	Add

Table 3.6: Overview of projects used in the correctness evaluation (2/2).

Graphing Calculator		
Ev.	**Feature model change**	**Type**
1	Function input	Add
	Coordinate system	Add
	Graphs	Add
2	Placed points	Add
	Table	Add
	Regression	Add
3	Placed points, Table → Placed points dataset, Table dataset	Split
	Regression → … from placed points, … from table	Split
	Variables	Add
Workout App		
Ev.	**Feature model change**	**Type**
1	Sessions	Add
	SessionForm	Add
	SessionHistory	Add
	Exercises	Add
	ExerciseForm	Add
	ExerciseList	Add
2	Exercises → StrengthExercises, CardioExercises	Split
	ExercisesForm → StrengthForm, CardioForm	Split
	ExercisesList → StrengthList, CardioList	Split
3	Statistics	Add
	StrengthProgress	Add
	CardioProgress	Add
	Filters	Add
Budget Tracker		
Ev.	**Feature model change**	**Type**
1	Transactions	Add
	Categories	Add
	Per-category spend limit	Add
	Data persistence	Add
2	Recurring transactions	Add
	Saving goals	Add
	Currency converter	Add
	Import from bank	Add
3	Recurring transactions → Salary, Subscriptions	Split
	Currency converter → Standalone, In-transaction	Split
	Import from bank	Remove

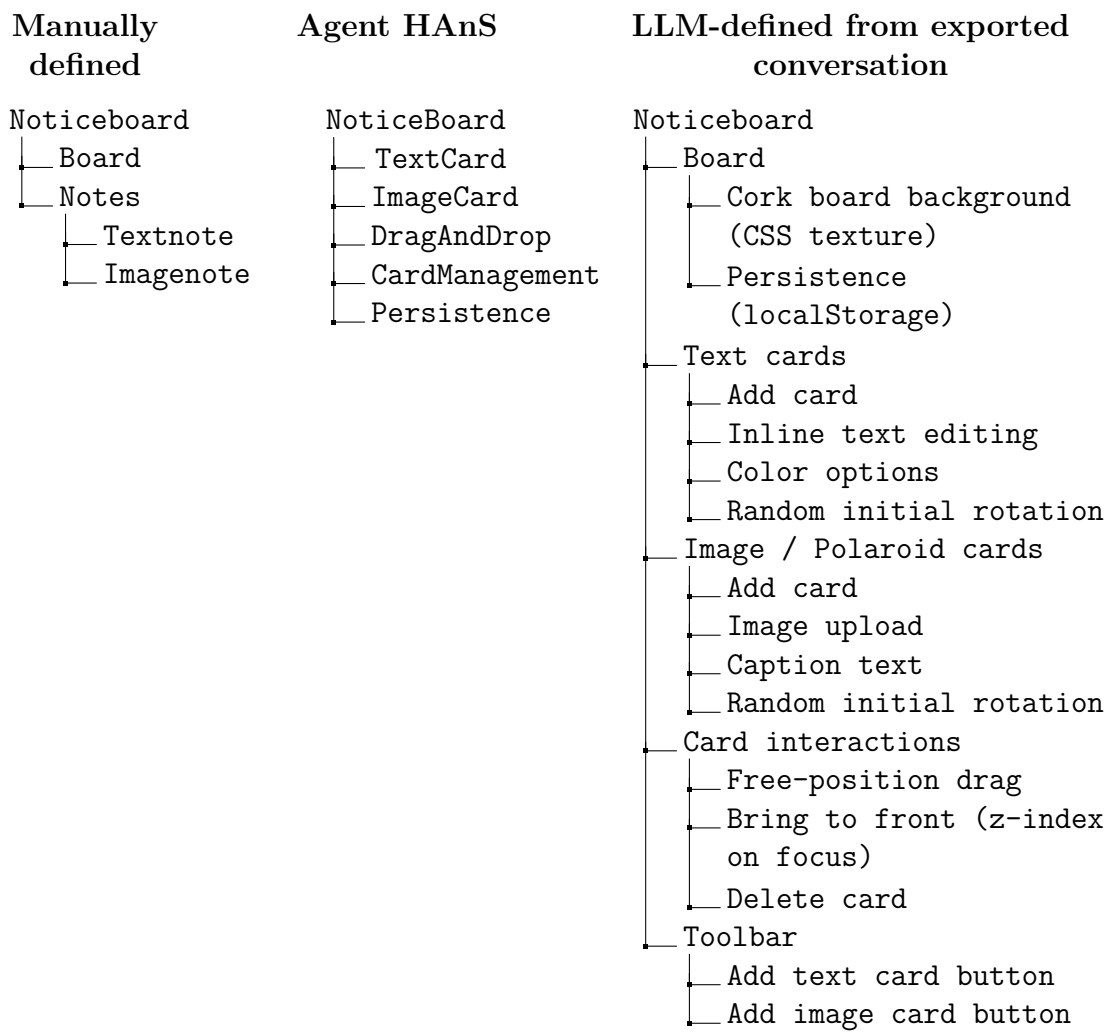


Figure 3.4: Three feature models generated for the Noticeboard project. Given prompt before evaluation point: *"Make a notice board React application. It should be a local webpage where notes and cards with text can be added and dragged around. There should also be image cards, like pinning a polaroid."*

The generated feature model...

1. ... is nonsense or non-existent
2. ... is syntactically valid, but irrelevant
3. ... is topically related, but not representative of the software
4. ... is relevant, but has one or more major flaws
5. ... is relevant, has no major flaws, but several minor flaws
6. ... is relevant, has no major flaws, and at most one minor flaw
7. ... matches expectation in content; structure differs but is semantically equivalent
8. ... matches manual expectation in content and structure
9. ... matches manual expectation and adds correct, relevant features beyond it

With the grading criteria scores can be given to the two generated feature models: (i) generated by Agent HAnS and (ii) from the post-hoc conversation extraction.

To give an example of what the grading process looks like, let's once again take a look at fig. 3.4. The first column houses the baseline the other two feature models will be compared against. First looking at the feature model generated by Agent HAnS we can see that the two different kinds of notes are present, but with the names *TextCard* and *ImageCard*. It has also produced a few extra features which are relevant. Still, the feature *Board* is missing, and as it is quite a critical feature for a notice board application it must be considered a major flaw. Having a major flaw leaves it at a score of 4.

Next, looking at the post-hoc extracted feature model we can see it is much larger and contains a much more granular set of features. It contains both the *Board* feature and the two types of notes, and a few additional features. Still, the amount of detail is not in line with the baseline. Since the additional granularity does not include any incorrect features, they are all considered as minor flaws. Several minor flaws without any major flaws results in a score of 5.

It's important to note that what constitutes a major and minor flaw, how naming discrepancies of the features are treated and other decisions in the grading process is largely up to personal interpretation.

4

Implementation of Agent HAnS

The implementation of Agent HAnS[1] consists of four main parts: (i) a global instruction file, (ii) skills, (iii) a MCP server and (iv) a graphical dashboard. Each will be described in more detail in the following sections.

4.1 Global Instruction File

The first challenge was to activate the workflow and enable the tool. There are many different kinds of entrypoints for LLM tools.

Include in prompt. One way to initialize a specific workflow is to include a trigger sentence inside a prompt. For example: *"Before moving on to the rest of this prompt, read the instructions in the skill file."* The disadvantage of this is that the user is required to insert this specific magic sentence at the beginning of every session. There is also the issue of remembering to do this each time.

Use specific agent. Using a specific agent with these instructions built in can mitigate the issues of repetitively having to input a specific sentence. The disadvantage of using an agent is that they are mutually exclusive. Multiple agents can't be active at the same time. This means a user might want to use another agent while still using the feature model workflow presented in this thesis, which wouldn't be possible with the agent solution.

Global instruction file. Agentic coding tools usually have ways to have a fixed instruction set that is automatically included in every prompt. This instruction set lives in a file called AGENTS.md when using OpenCode, and in CLAUDE.md when using Claude Code. The advantages of the global instruction file are that it relieves the user of remembering to include specific instructions in their prompts, and also enables the user to use whichever agent they prefer. A disadvantage is that the user might not want to use the feature model workflow all the time. This can be mitigated by placing the file in the project root rather than the global context, only enabling it on a per project basis.

We chose to use the global instruction file as the entrypoint as the other options

[1] `https://github.com/isselab/Agent-HAnS`

were not in line with our requirements.

In short the global instruction file has the following contents:

- Check that a git repository is present (required for the graphical dashboard).
- Call the MCP-server *get-feature-model* tool (endpoint). This deterministically gets the current feature model into the LLM's context.
- Use the feature model skill to decide how the feature model will update, based on the context of the current conversation.
- Use the embedded feature annotations skill to prepare how to write annotations before moving on to write the actual code.
- Write the code and verify that written annotations are present in the feature model.
- Call the MCP-server *summary-gui* tool to summarize all changes to the feature model and display the web graphical dashboard to the user.

The instruction file contains precise and emphatic language to ensure it does what it's asked and doesn't accidentally miss any steps.

4.2 Skills

The agent does not by itself know how to create a feature model in the correct format or the syntax for the embedded feature annotations. Therefore the tool uses two skills which instruct the agent on how to work with embedded feature annotations and the feature model.

Feature model skill. The feature model skill instructs the LLM on the correct format and structure of a feature model. It specifies that the feature model must be stored in a `.feature-model` file in the project root, that feature names must be written in Pascal case, and that hierarchy is expressed through indentation. The skill also provides guidance on how to reason about feature relationships, such as distinguishing between features that are hierarchically dependent and those that are independent but cross-cutting.

Embedded feature annotation skill. The embedded feature annotation skill instructs the agent on the correct syntax for embedding feature annotations in source code. It defines the different annotation forms and gives examples on how to use them. The skill further specifies that annotations must be placed inside comments, that feature names must be in Pascal case, and that whole-file feature mappings should be expressed using a `.feature-to-file` file rather than wrapping an entire file in annotations.

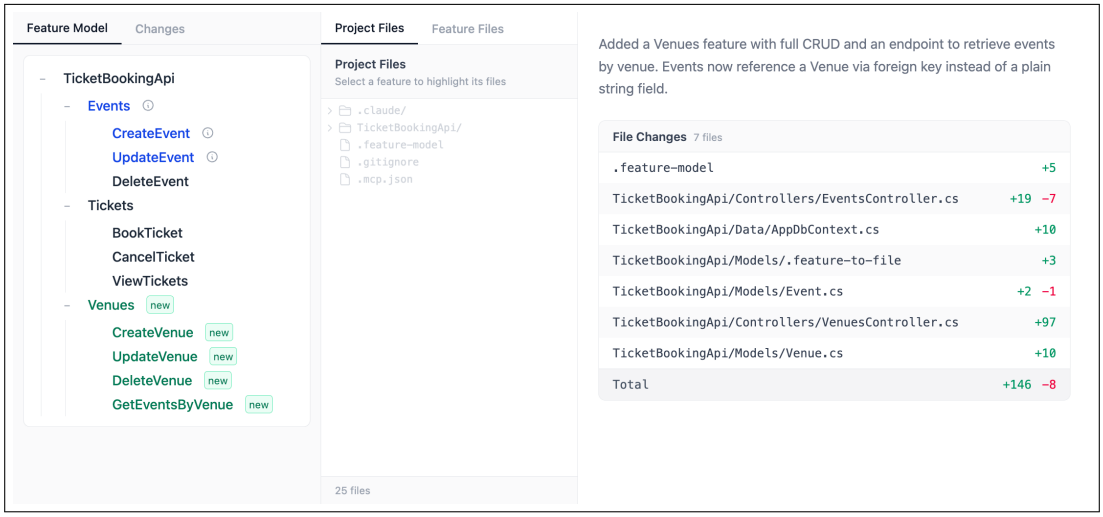


Figure 4.1: The dashboard displaying a summary of changes made to the example project *TicketBookingApi*.

4.3 MCP Server

The MCP server acts as a bridge between the agent and components not part of the LLM-tool. It also allows the agent to call pre-defined methods which produce deterministic outcomes. The MCP server exposes two tools: *get-feature-model* and *summary-gui*.

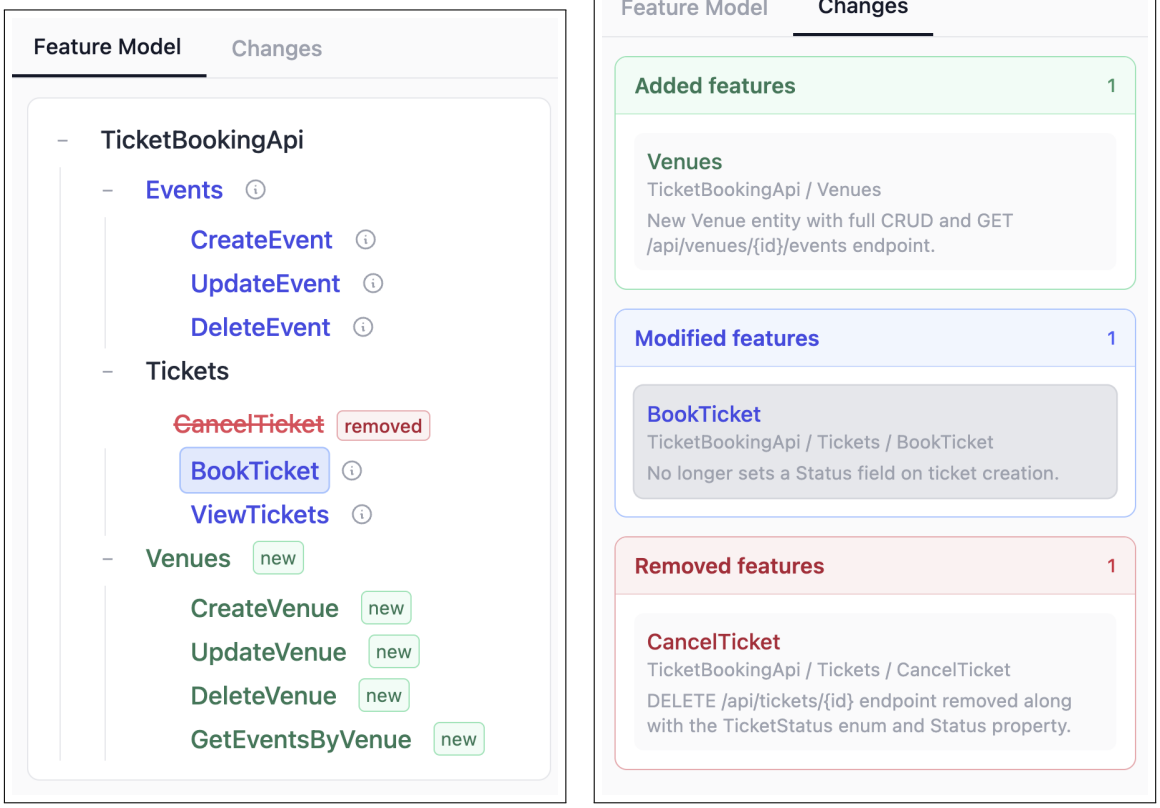
get-feature-model. The get feature model tool supplies the agent with the current feature model in a deterministic way. Rather than relying on the LLM to discover or infer the feature model from the codebase, this tool ensures that the LLM always receives the current feature model, reducing the risk of inconsistencies.

summary-gui. The *summary-gui* tool is invoked by the agent at the end of a prompt response. The tool begins by retrieving the relative difference using Git's *diff* function to identify which files have been changed, then inspects the embedded feature annotations to associate features with the files they are present within. From this, it constructs a file tree reflecting the changes and their feature mappings, and aggregates all collected data into a JSON structure which is persisted in memory under a unique identifier. Finally, a browser window is opened to display the graphical dashboard.

4.4 Graphical Dashboard

The graphical dashboard consists of a locally hosted web application that is served by a web server integrated into the MCP server. An example of what the dashboard looks like with a summary displayed can be seen in fig. 4.1.

The dashboard is composed of the following views:



(a) The feature model view with the *BookTicket* feature selected.

(b) The changes view displaying three changes. Note that some changes were hidden in order improve visibility.

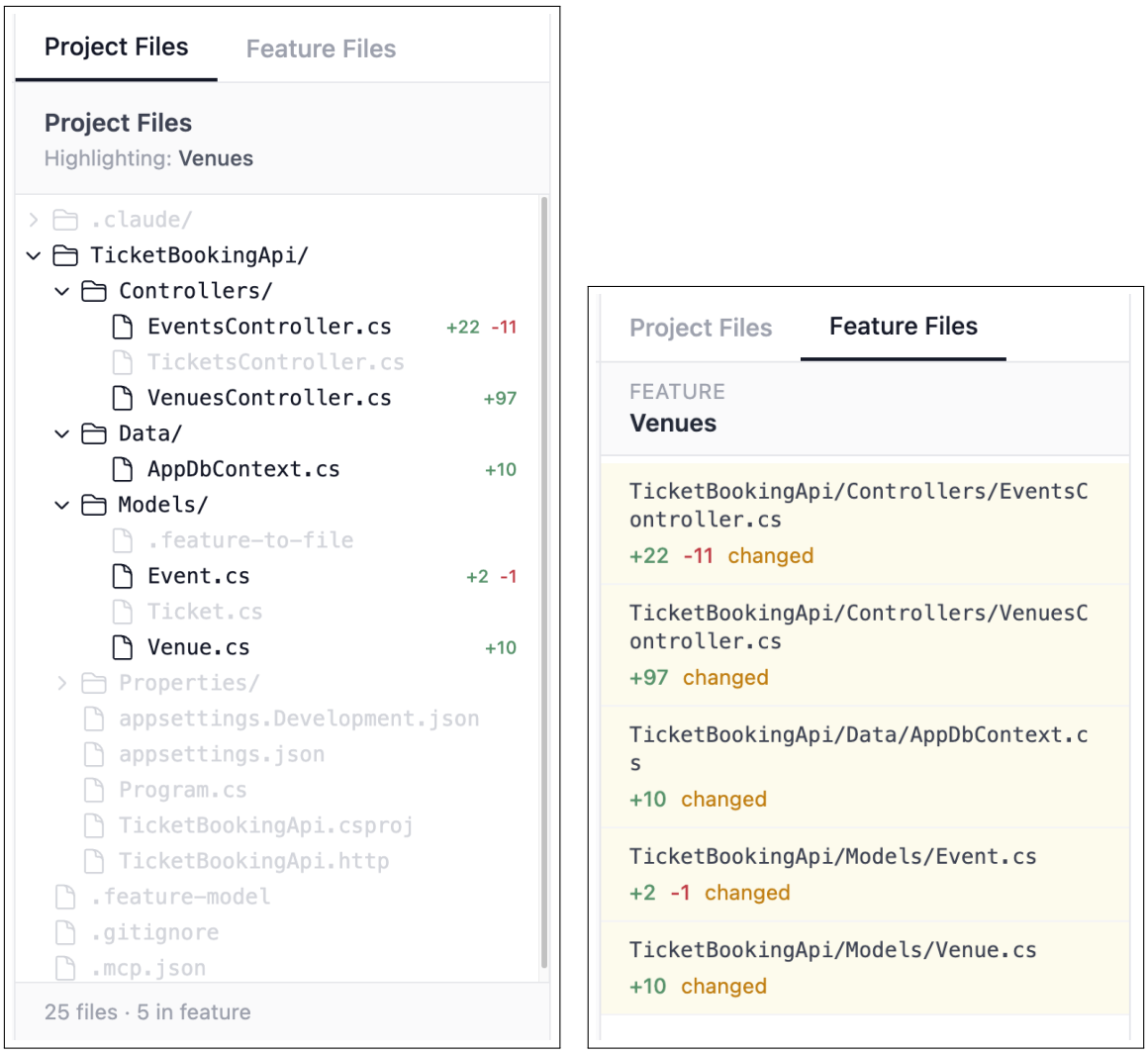
Figure 4.2: The feature model and changes views.

Feature Model. The feature model view is an interactive tree view of the feature model. There are four kinds of features: (i) unchanged features without color, (ii) added features in green, (iii) changed features in blue, and (iv) removed features in red. Each modified or added feature can be expanded to show a summary of the change made. By selecting a feature, related files are highlighted in the project files and feature files views. The feature model view can be seen in fig. 4.2a.

Changes. The changes view is a list of summary cards sorted into three sections: (i) added features in green, (ii) changed features in blue, and (iii) removed features in red. Each card contains a summary of the changes to the feature. The changes view can be seen in fig. 4.2b.

Project Files. The project files view shows a tree of the project's files. Each changed file displays a plus and minus count indicating added and removed lines. Selecting a feature in the changes or feature model view highlights related files. Selecting a file opens a file diff. The project files view can be seen in fig. 4.3a.

Feature Files. The feature files view lists all files associated with a selected feature. Files changed during the latest prompt are marked as changed with added



(a) The project files view with files related to the *Venues* feature highlighted.

(b) The feature files view focused on the *Venues* feature.

Figure 4.3: The project files and feature files views.

and removed line counts. Selecting a file opens a file diff. The feature files view can be seen in fig. 4.3b.

File Changes. The file changes view lists all files modified in the project, giving the user an overview of the scope of changes made. The file changes view can be seen in fig. 4.4.

File Diff. When a modified file is selected in the feature files or project files view, its full contents are shown with changed lines highlighted, providing a detailed view of exactly what was modified. The file diff view can be seen in fig. 4.5.

File Changes 7 files

.feature-model	+5
TicketBookingApi/Controllers/EventsController.cs	+19 −7
TicketBookingApi/Data/AppDbContext.cs	+10
TicketBookingApi/Models/.feature-to-file	+3
TicketBookingApi/Models/Event.cs	+2 −1
TicketBookingApi/Controllers/VenuesController.cs	+97
TicketBookingApi/Models/Venue.cs	+10
Total	+146 −8

Figure 4.4: The file changes view showing all modified project files.

MODIFIED **TicketBookingApi/Controllers/TicketsController.cs**

```
diff --git a/TicketBookingApi/Controllers/TicketsController.cs b/TicketBookingApi/Controllers/Tickets
index 8dfb925..1e5d1b2 100644
--- a/TicketBookingApi/Controllers/TicketsController.cs
+++ b/TicketBookingApi/Controllers/TicketsController.cs
@@ -22,7 +22,7 @@ public class TicketsController(AppDbContext db) : ControllerBase
             {
                 t.Id, t.EventId,
                 EventName = t.Event.Name,
-                t.HolderName, t.HolderEmail, t.BookedAt, t.Status
+                t.HolderName, t.HolderEmail, t.BookedAt
             })
             .ToListAsync();
         return Ok(tickets);
@@ -49,8 +49,7 @@ public class TicketsController(AppDbContext db) : ControllerBase
             EventId = request.EventId,
             HolderName = request.HolderName,
             HolderEmail = request.HolderEmail,
-            BookedAt = DateTime.UtcNow,
-            Status = TicketStatus.Active
+            BookedAt = DateTime.UtcNow
         };

         ev.AvailableTickets--;
```

Figure 4.5: The file diff view showing changes within *TicketsController.cs*

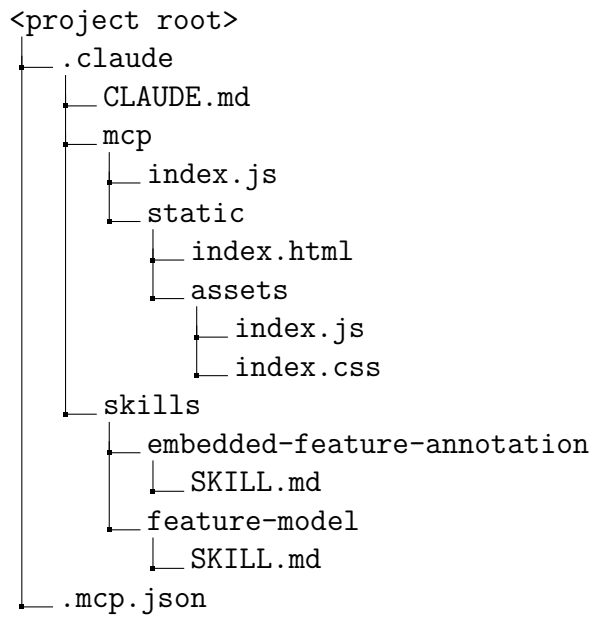


Figure 4.6: Directory tree of all files included in the tool.

4.5 Usage and Installation

The workflow can be added to a project by adding a file and a folder to the project root, as shown in fig. 4.6.

With Agent HAnS installed the prompting process is by intention unaffected. The user writes prompts as usual, with any agents, MCP-servers and plugins they prefer. The tool will read the feature model if it is present, otherwise it will create a new one. It will then modify it according to the given prompt. It then starts writing code, which includes embedded feature annotations in line with the modified feature model. With the exception of the embedded annotations, code generation is unmodified. Once it has finished generating code it calls the MCP server's summary tool. In the web-based dashboard the user can explore the changes made by their last prompt. After exploring the changes made they are free to return to the AI coding tool and write their next prompt. Figure 4.7 shows a visualization of the usage flow.

If Agent HAnS is used inside an existing repository without a feature model and embedded feature annotations, the tool will function to some degree, but it has not been designed for it. While it will annotate code and define a feature model for parts relating to the prompt, it will not, however, scan the entire source code to extract a complete feature model and write annotations throughout.

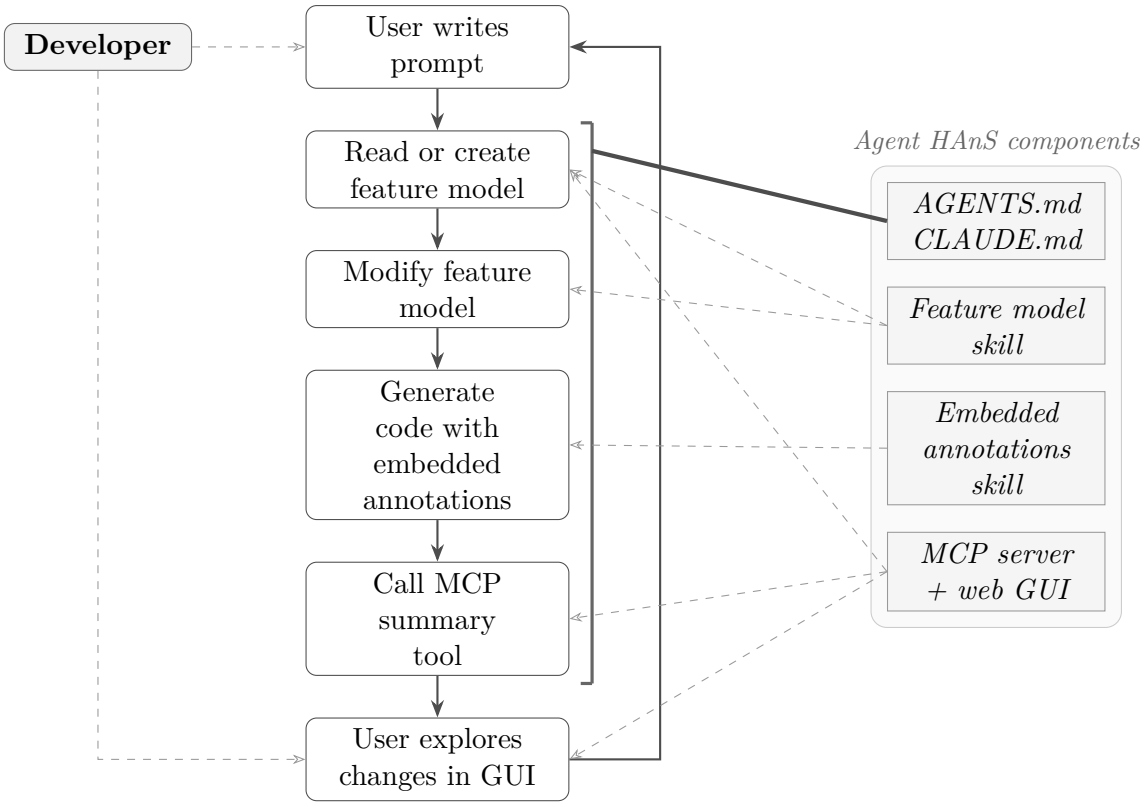


Figure 4.7: Usage flow of Agent HAnS and the involved components.

5

Results

This chapter presents the findings for each of the research questions. Section 5.1 addresses RQ 1 by presenting how Agent HAnS has evolved over the DSR cycles. Section 5.2 addresses RQ 2 by presenting the result of the user study and the correctness evaluation.

5.1 RQ 1: The Artifact

While chapter 4 contains a complete description of the final version of Agent HAnS, this section describes how Agent HAnS evolved over the cycles and how it meets the requirements and guidelines defined in chapter 3. Table 5.1 shows how each cycle meets each of the defined guidelines.

5.1.1 First Cycle

In the first cycle, the result was a simple artifact which acted as a prototype and discussion material. It was a complete workflow that included a global AGENTS.md file which describes the high level workflow, and also ensures that the workflow happens without any specific action required from the developer. In the next step a subagent is called that identifies features and inserts them into the feature model based on a description included in its file. The main agent then utilizes a skill to add embedded feature annotations to the generated code in accordance with a style guide and the earlier modified feature model.

Table 5.1: How each cycle meets each of the defined guidelines.

Guideline	Cycle 1	Cycle 2	Cycle 3
1 Display the feature model visually	✓	✓	✓
2 Map features and files	×	✓	✓
3 Maintain a feature model	✓	✓	✓
4 Generate embedded annotations	✓	✓	✓
5 High compatibility	✓	✓	✓
6 Integrate with existing workflows	✓	✓	✓
7 Address comprehension challenge	*Addressed in RQ 2*		
8 Address trust challenge			

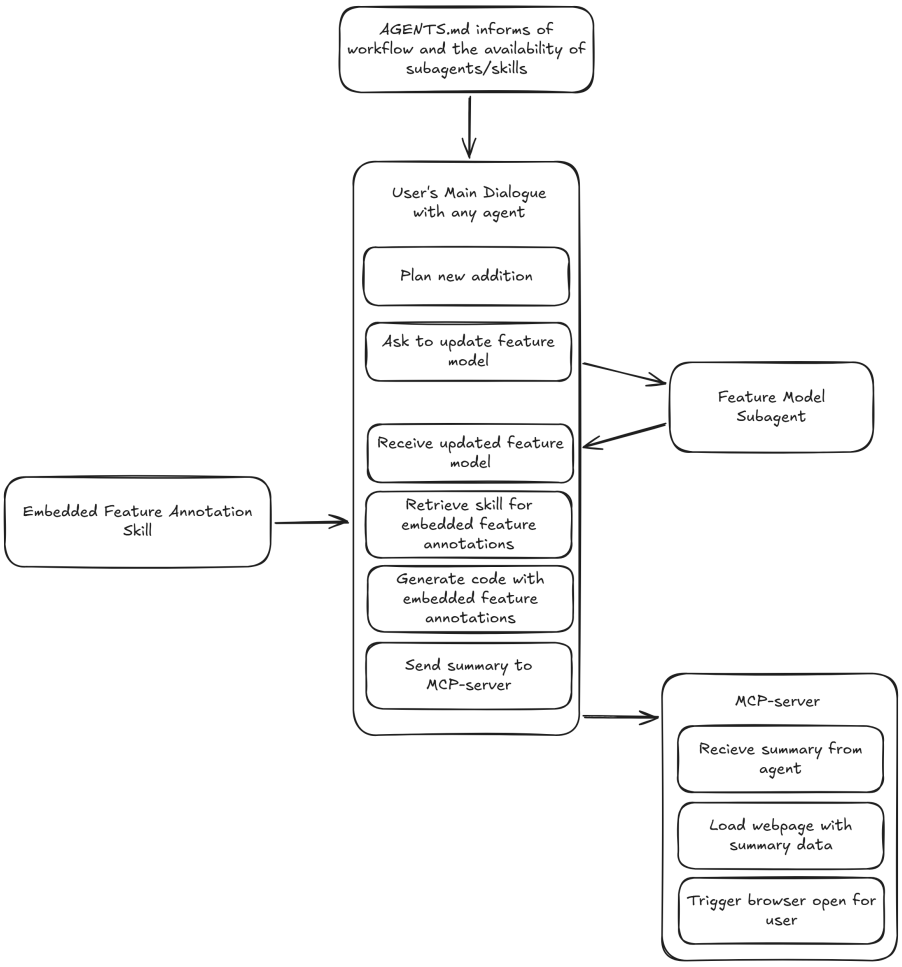


Figure 5.1: Diagram describing the flow of the first cycle prototype.

Finally the LLM connects to an MCP server that asks for a summary of changes made to the feature model and itself. The summary is sent to a precompiled static webpage that displays the feature model with descriptions for added, modified and removed features. The webpage is opened automatically and the developer is free to explore the changes and the feature model. Once done they can return to their dialogue and the webpage will update the next time the feature model is updated. The complete flow is visualized in fig. 5.1.

From the defined guidelines the first prototype does not include an interface for showing relations between features and files. It does not make use of the generated embedded feature annotations. Therefore guideline 2 is not met by the first cycle prototype.

The other guidelines are met as the webpage shows the feature model visually (#1). The subagent and the skill ensures an updated feature model and embedded feature annotations (#3, #4). Using a global instruction file and MCP-server, instead of a plugin enables Agent HAnS to be used together with many different tools, without interfering with the typical LLM-based coding workflow drastically (#5, #6).

5.1.2 Second cycle

The core ideas remained for the second cycle with the artifact being a summary of what code had been generated since the last prompt and giving the developer insights into how the feature model has changed. The feature model subagent was changed to instead being a skill as the independence of it being a subagent was not utilized by the language models. The instruction files were also revised to handle edge cases and give clearer instructions for the LLMs, for example what to do when a feature model isn't already present.

Feature wise a file tree was added showing all files in the project. When clicking on a feature in the feature model each file which contains code for that feature is highlighted giving the developer an idea of where a features code is located. This ensures guideline 2 is met. None of the other guidelines regressed as part of the evolution during the second cycle.

Beyond this, a Git-style diff-summary was also added, under the reasoning that it can help both comprehension and trust seeing what parts have been changed in more detail. It shows what files have been changed since the last commit, helping the developer get insights into how the code and file structure evolves over time. This can be helpful to see that expected files are affected, or hint at wrongful execution where files not belonging to a modified feature are affected by the LLM.

5.1.3 Third cycle

The complete description of the final implementation is shown in chapter 4. During the final cycle none of the met guidelines regressed, and as the focus was more towards evaluation of Agent HAnS the changes to the implementation were minimal. The changes made in the global instruction file were clarifications like ensuring a Git repository is initialized and changes to wording to make the language more emphatic and precise.

5.2 RQ 2: Evaluation of Agent HAnS

The evaluation of the tool was performed in two main dimensions. One for the objective correctness of the feature models being generated. The other is for the benefits experienced by users of the tool in a user study.

5.2.1 User Study

Code comprehension. To measure the tool's effect on code comprehension and address RQ 2.1, participants answered questions about the code base after each step of the tasks. Table 5.2 presents the number of participants who successfully answered the code comprehension questions for task 1, while table 5.3 presents the corresponding results for task 2.

Looking into the details some of the incorrect answers show that they do not have

Table 5.2: Number of participants who provided correct answers to the comprehension questions for task 1. Each group had seven (7) participants.

Question	Without tool Group 2	With tool Group 1
If you needed to add a "Brunch" filter option, what would need to change?	6	5
Did the LLM put search and filter in a single component or keep them in separate files rendered side by side?	7	6
If you needed to add a "Mocktail" category for non-alcoholic drinks, what would need to change?	6	7
If a recipe's ingredient list is edited after its ingredients were already imported, does the shopping list reflect the change?	6	7
Total	25 (of 28)	25 (of 28)

to do with comprehension of the software in relation to the code being generated by AI, but due to other factors. To the question *Is the modal a separate component or inline conditional rendering?* one incorrect answer was *"I think it is inline. Sorry I'm not a frontend developer."*.

NASA-TLX comparison. To measure how the tool affects the overhead and required effort from developers, and to partially answer RQ 2.3, a NASA-TLX survey was included. Note that only the raw NASA-TLX scores were collected and are presented without weights. NASA-TLX scores range from 0 to 100, where 100 indicates a high workload.

The following six questions were included in the survey after each task:

- How mentally demanding was the task?
- How physically demanding was the task?
- How hurried or rushed was the pace of the task?
- How successful were you in accomplishing what you were asked to do?
- How hard did you have to work to accomplish your level of performance?
- How insecure, discouraged, irritated, stressed, and annoyed were you?

Table 5.4 shows average scores per subscale broken down by tool access. The overall averages are nearly identical: 22.1 without the tool and 21.6 with it ($p = 0.86$, $d = -0.05$), suggesting no meaningful difference in aggregate workload. At the subscale level, the only statistically significant difference was physical demand, which was higher with tool access ($M = 13.6$ with, $M = 5.7$ without). Mental demand and frustration were both lower with tool access ($d = -0.31$ and $d = -0.42$ respectively),

Table 5.3: Number of participants who provided correct answers to the comprehension questions for task 2. Each group had seven (7) participants.

Question	Without tool Group 1	With tool Group 2
If you needed to add a fourth priority level, what would need to change?	6	7
Were any existing files modified when replacing due dates with deadlines, or was the change purely additive?	6	5
Is the modal a separate component or inline conditional rendering?	7	6
Did the LLM merge the existing implementations for deadline and priority or create an entirely new solution for urgency?	6	7
Total	25 (of 28)	25 (of 28)

Table 5.4: Average raw NASA-TLX scores by subscale and condition, with paired t-test p-values and Cohen's d effect sizes (n = 14).

	Without tool		With tool			
Subscale	**M**	**SD**	**M**	**SD**	**p**	**d**
Mental	34.3	27.7	26.4	17.8	0.27	−0.31
Physical	5.7	12.8	13.6	18.2	0.01	+0.81
Temporal	28.6	25.5	32.9	25.5	0.60	+0.14
Performance	20.7	14.4	20.7	18.6	1.00	+0.00
Effort	22.1	19.3	22.9	17.3	0.83	+0.06
Frustration	21.4	20.3	12.9	13.3	0.14	−0.42
Total	22.1	17.6	21.6	14.8	0.86	−0.05

suggesting a small effect, though neither reached statistical significance ($p = 0.27$ and $p = 0.14$). All remaining subscales showed negligible differences.

Table 5.5 and fig. 5.2 break scores down by task and group, where shaded columns indicate tool access. A notable difference is seen between the two groups: group 1 reported consistently higher workload across both tasks (average 25.8) compared to group 2 (average 17.9). This gap persists regardless of tool access, which suggests inherent differences between how the groups perceive workload. The implications of this are discussed further in chapter 6. The difference between the two tasks is minimal, with task 1 having an average of 22.3 and task 2 21.4.

Domain-specific questions. Table 5.6 and fig. 5.3 show average scores per domain-specific statement broken down by tool access. Overall, differences were small across all items, with no result reaching statistical significance. The largest effect was on trust that relevant parts of the codebase were addressed, which was slightly higher with Agent HAnS (M = 3.6 with, M = 3.4 without, p = 0.41, d = +0.23), suggesting a small positive trend. Understanding of code structure

Table 5.5: Average raw NASA-TLX score by task and group. Grey columns mark tool access.

	Task 1				Task 2			
Subscale	**Group 2**		**Group 1**		**Group 1**		**Group 2**	
	M	*SD*	*M*	*SD*	*M*	*SD*	*M*	*SD*
Mental	20.0	21.6	32.9	17.0	48.6	26.7	20.0	17.3
Physical	0.0	0.0	18.6	22.7	11.4	16.8	8.6	12.1
Temporal	35.7	28.8	37.1	28.1	21.4	22.7	28.6	24.1
Performance	21.4	12.1	22.9	22.1	20.0	17.3	18.6	15.7
Effort	18.6	20.4	28.6	16.8	25.7	19.0	17.1	17.0
Frustration	15.7	24.4	15.7	15.1	27.1	15.0	10.0	11.5
Total	18.6	13.0	26.0	15.2	25.7	11.4	17.2	10.1

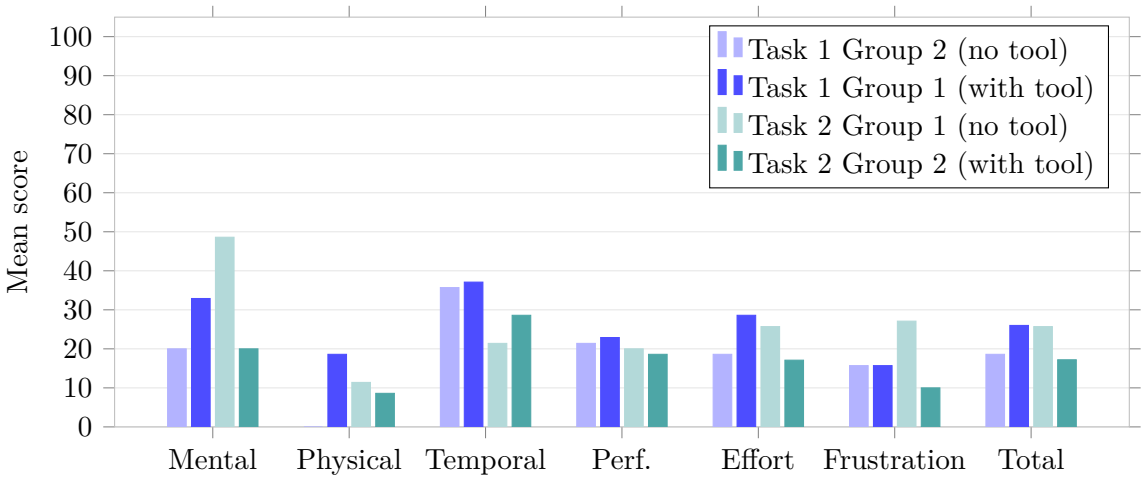


Figure 5.2: Bar plot of the NASA-TLX subscales. Color marks tasks, the darker color is the treatment with Agent HAnS. Group 1 are the two inner bars, group 2 is the two outer bars for each subscale.

Table 5.6: Average domain-specific question scores by condition, with paired t-test p-values and Cohen's d effect sizes. All items rated on a 1–5 Likert scale (n = 14).

	Without tool			With tool				
Statement	**M**	**Med**	**SD**	**M**	**Med**	**SD**	**p**	**d**
I have a good understanding of what the generated code looks like and how it is structured	3.6	3.5	1.2	3.4	3.5	0.9	0.22	−0.35
I could have explained the changed code to another developer	3.4	3.0	0.9	3.4	3.0	0.9	1.00	+0.00
I felt uncertain about what was happening in the codebase at some point during the task	3.1	3.0	1.1	3.1	3.5	1.4	0.84	+0.06
I trust that all parts of the codebase relevant to my request were addressed	3.4	4.0	0.8	3.6	3.0	1.0	0.41	+0.23
I knew where I would look if I wanted to verify the changes that were made	3.8	4.0	0.9	3.9	4.0	1.0	0.64	+0.13
I would feel confident handing this codebase off to another developer	2.9	3.0	1.0	2.9	2.0	1.2	1.00	+0.00

was slightly lower with Agent HAnS (M = 3.4 with, M = 3.6 without, p = 0.22, $d = -0.35$), suggesting a small negative trend. This difference is unlikely to be meaningful given the effect size and lack of significance. The remaining statements showed negligible differences in effect.

Usability of the tool. Each group had access to the tool for one of the tasks, and after that task they were asked about the usability of the tool in the form of a System Usability Scale (SUS) survey. These results will be used to answer RQ 2.3. The total average was 73.9, with a standard deviation of 10.5. This score is above the threshold of 70, as defined during the design of the study. five of the 14 participants gave a SUS score of below 70. six of them gave scores of 80 or above. The averages between the groups differed minimally: 73.2 and 74.6 for Group 1 and 2 respectively.

The standout individual statements from the SUS survey were three statements relating to the ease of use, as they received high average scores (the maximum possible average is 5):

- *I thought the system was easy to use* (avg. 4.4),
- *I think that I would need the support of a technical person to be able to use this system* (inverted avg. 4.4),

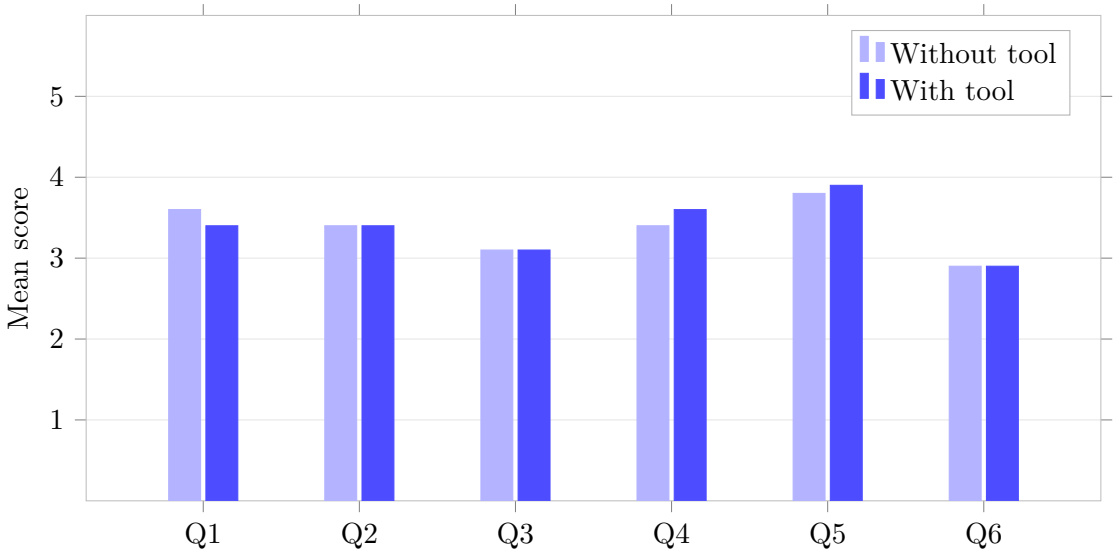


Figure 5.3: Bar plot of the mean results of the domain-specific questions.

Table 5.7: Results of questions about how the tool handles feature models. Likert scale, 1-5.

Statement	Average		SD
The features shown in the feature model felt relevant and of quality	4.1	Agree	0.5
There was code that belonged to a feature that was not annotated	2.3	Disagree	0.9

- and *I needed to learn a lot of things before I could get going with this system* (inverted avg. 4.6).

The lowest performing question was the first question: *I think that I would like to use this system frequently* with an average of 2.8, the only question to receive a below neutral score of 3.

Perceived Feature Model Correctness and Relevancy. In addition two questions specific to the feature model domain were asked, about how well the participants thought the tool performed in relation to the feature model and the embedded feature annotations, as seen in Table 5.7. This gives a user's qualitative perspective on RQ 2.2, as an addition to the correctness evaluation.

Qualitative Feedback. At the end of the survey four open-ended questions were asked.

To the question *Did you experience any challenges using the tool?* the responses were mostly positive, with a few personal challenges like this example from participant 2: *"At first I was running Claude in Docker, and that prevented it from using the MCP server, so I had to run it locally instead."*.

Table 5.8: Evaluation point scores by project for the correctness evaluation.

	Tool			Post-hoc extraction		
	Ev. 1	**Ev. 2**	**Ev. 3**	**Ev. 1**	**Ev. 2**	**Ev. 3**
Battleships	6	6	6	6	5	4
Timer	4	4	4	4	4	5
Graphing calculator	8	8	6	5	5	5
Notice board	4	4	4	5	5	5
Budget tracker	8	7	7	5	5	5
Workout app	4	4	4	4	4	4

To the question *What did you like about using the tool?*, participants mentioned the overview of changes and ease of use as benefits of the tool. The notion of usefulness in a larger scale project was also brought up by participant 8: *The summary when completed with the task gave a very good overview of what was changed/deleted/added. Could be especially important when doing large scale changes.*

To the question *Any suggestions to improve the tool in the future?*, most participants did not leave a comment, but the few that did noted that improvements could be made to the way the files are shown and the integrated file diffs. A dark mode was also requested by participant 3.

A final catch-all question was included where the participants could give other comments or give nuance to the other questions in the user study. One participant noted that using Agent HAnS seemed to increase the time of the agent to complete its responses. Another participant left the following quote: *"The functionality of the tool did not provide me with a lot of value compared to me just reading the code, in an IDE, or in Claude Code. Maybe for bigger more complex tasks, but for this, I didn't feel the added value."*.

5.2.2 Correctness Evaluation

The results from the second benchmark can be seen in table 5.8. The two columns, one for feature models generated by Agent HAnS and the other for the post-hoc extraction, list the scores across the three evaluation points for each project. Each project is scored in three steps, where each step can get a score from 1-9. How each score was defined can be found in chapter 3. The median result for the feature models generated by Agent HAnS was a split between 4 and 6. For the post-hoc extraction the median result was 5.

6

Discussion

This chapter discusses the results of the study in relation to the research questions. Section 6.1 reflects on the design and implementation of Agent HAnS, addressing RQ 1. Section 6.2 discusses the effectiveness of the tool in terms of developer comprehension and trust, feature model correctness, and usability, addressing RQ 2. The chapter concludes with a discussion of threats to validity and future work.

6.1 Implementation of the Tool (RQ 1)

How can a tool be designed that leverages feature models and embedded feature annotations in order to make LLM-based coding techniques more comprehensible and trustworthy?

The implementation of Agent HAnS is based around the guidelines defined in section 3.1.1. After the final cycle guidelines 1-6 are met, see table 5.1 (#7 and #8 are discussed in section 6.2), and Agent HAnS is a fully working support tool that offers a continuously maintained feature model, change summaries and feature to file locations through embedded feature annotations. Agent HAnS can work together with many different AI coding tools thanks to the universality of the Model Context Protocol (MCP). The ideas for how to further develop Agent HAnS is presented in section 6.4.1.

During the development of Agent HAnS and from the participants of the user study the general overview and the clarity of the changes made were brought up as appreciated aspects, see section 5.2.1. The participants did not have any major challenges using Agent HAnS with only small suggestions for improvement, like adding a dark mode.

Whether Agent HAnS meets guidelines 7 and 8, the challenges regarding comprehension and trust, is discussed in the following sections. The other challenges listed in appendix C were not considered when designing the tool. Under the premise that feature models were to be involved the choice of which of the identified challenges to address came naturally. One of the reasons for including feature models and embedded feature annotations in a project is because it eases understanding of the codebase on a high level [31]. From the interviews we also gathered that the rea-

son developers didn't trust the generated code was often due to a lack of perceived understanding. Understanding and trust were natural inclusions, but when looking at the rest of the list none of the other challenges had a clear fit. Challenges listed under *Limitations from providers* have more to do with the companies offering the LLMs. The other challenges under *User familiarity*, like the challenge of *Writing descriptive prompts*, are addressable but through other means than feature models.

Still, some of the challenges while not directly the aim of Agent HAnS or the thesis, can benefit from using Agent HAnS. For example, a pair of challenges under *How models act* is that a response can either be too narrow, meaning it misses some of the expectation, or too broad, meaning it makes changes beyond the given instructions. With Agent HAnS it is possible to quickly see if a feature is missed or included when it was not supposed to. While it does not directly mitigate incorrect generation, it makes the developer aware of the problem meaning they can correct the course with their next prompt.

Agent HAnS addresses guidelines 1–6 set at the beginning of the thesis and we have shown how a comprehension supporting LLM-tool can be designed.

Conclusion

6.2 Effectiveness of Agent HAnS (RQ 2)

How effective is the tool in supporting developers during LLM-based coding?

The answer to RQ 2 is the combined answers for its subquestions in the following sections.

6.2.1 Mitigation of Challenges (RQ 2.1)

How does the tool affect developers' understanding and trust of LLM-generated code?

The core of this thesis was using feature models as a tool to improve the challenges faced by developers in LLM-based code generation. Of all the identified challenges described in chapter C only two were selected as being able to be targeted by the tool. The two categories *Limitations from providers* and *How models act* were mostly out of our control. Both user understanding and trust looked to be both achievable to improve with a tool, and was some of the most recurring challenges lifted during the interviews.

To answer if Agent HAnS improves a developer's understanding and trust of LLM-generated code domain-specific statements were posed to the participants after each task, with the results presented in table 5.6. Overall the differences between conditions were small and none reached statistical significance.

The trust in changes to the codebase showed the largest positive trend ($d = +0.23$),

while understanding of code structure showed a negative trend with tool access ($d = -0.35$), though neither result is likely meaningful given the lack of significance and small sample size. A speculative explanation for the negative trend in understanding could be that participants unfamiliar with feature models perceive an increased complexity having to think about one additional concept. We also asked comprehension related questions after each task step, with the results from the questions in table 5.2 and table 5.3. The aggregated results between the two groups did not differ. The few incorrect answers were evenly distributed across both groups, suggesting they represent baseline noise rather than any systematic effect. A conclusion that can be drawn from this is that Agent HAnS does not have a measurable effect on understanding and trust in the context of this user study.

A reason for this could be that the tasks used in the user study were not large or complex enough to elicit the challenges of comprehension and trust, which Agent HAnS intends to mitigate. Due to the inherent time limitations set by the thesis each task was only allowed up to an hour. Within this time-frame the projects never became large enough for the developers to lose the mental grasp of the generated code, independent of the presence of Agent HAnS. If the study was significantly longer, following developers over weeks or months with larger, more complex projects, the perceived challenges may become more pronounced and in turn make any potential effects easier to observe. This was also noted by one of the participants of the user study, who wrote in one of the open-ended questions: *"The functionality of the tool did not provide me with a lot of value compared to me just reading the code, in an IDE, or in Claude Code. Maybe for bigger more complex tasks, but for this, I didn't feel the added value."*

As neither the domain statements or the comprehension questions in the user study show a significant difference when using Agent HAnS it can not be concluded that Agent HAnS provides a benefit to comprehension or trust, at least not with the scale of the performed user study.

Conclusion

6.2.2 Correctness (RQ 2.2)

What is the quality of LLM generated feature models and embedded feature annotations?

The results for the correctness evaluation are presented in table 5.8. Each feature model was scored on a non-linear scale from 1–9, where a score of 8 indicates performance that matches or is equivalent to the baseline feature model, as described in chapter 3.

The results show that the feature models generated by Agent HAnS are evenly split into two groups depending on the project, with a median split between 4 and 6. For half the projects' evaluation points (timer, notice board and workout) it only

achieves a score of 4, but for the three other projects the scores are higher. The feature models from the post-hoc extraction are more consistent where both the median and mode are 5. For most of the extracted feature models the amount of features included and the granularity of the features were high compared to the baseline model (fig. 3.4 for reference). As the over-inclusion of detail wasn't present in the baseline, but not strictly incorrect, it only counted as several minor flaws. This correlates with the mode score of 5.

Agent HAnS tends to either match the baseline closely or miss a critical feature, resulting in two distinct clusters of scores. The post-hoc extracted feature models avoid this pattern by consistently over-including features rather than risking omission. This results in a smaller score deduction as the flaws are only considered minor. This explains both the higher consistency of the post-hoc models and why Agent HAnS scores higher when it doesn't make a critical mistake.

An important note on the baseline feature models is that they are manually defined by us, and our experience of creating feature models is limited. This resulted in a few cases where a generated feature model was perceived as better than the baseline defined by us, but in turn received a lower score because the suggested features did not align with the pre-defined baseline. How this affects the results is discussed in threats to validity, section 6.3, and ways to improve a similar correctness study is discussed in future work, section 6.4.3.

In addition to the correctness evaluation two statements were posed in the user study related to the quality of the feature model and the embedded feature annotations, as seen in table 5.7. These results complement the results form the correctness benchmark being more qualitative in nature. We can see that the participants agreed that the feature models generated by Agent HAnS felt relevant and of quality, and that they did not find that feature-belonging code was left unannotated. It is not clear whether the reason that the feature model relevancy score isn't higher is due to participants being uncertain what a relevant feature model looks like, or if they observed feature models which weren't in line with their expectations.

We can conclude that the feature models generated by Agent HAnS varies from being equivalent to a human defined baseline to missing a critical feature. Compared to feature models generated without it, Agent HAnS returns feature models that are closer to a human defined baseline, as long as it doesn't make a critical mistake. Users of Agent HAnS find the generated feature models to be relevant and of quality.

Conclusion

6.2.3 Usability (RQ 2.3)

How do the users of the tool perceive its usability?

From the user study both the NASA-TLX scores and the results from SUS survey give a picture of the usability of the tool. Overall NASA-TLX scores were nearly

identical across conditions (22.1 without, 21.6 with, p = 0.86, $d = -0.05$), suggesting the tool does not meaningfully increase or decrease aggregate workload. The one exception was physical demand, which was significantly higher with tool access (p = 0.01, d = +0.81). This could be a result of Agent HAnS introducing an additional graphical dashboard to navigate after each prompt, adding physical interaction that would not otherwise occur. Another plausible explanation is that the physical demand subscale was confusing to the participants for a task with overall little physical exertion required. It it important to note that the absolute difference is only 8 points, which is similar to mental demand discussed next, but a bigger significance is shown as the relative difference is much greater. Mental demand and frustration both trended lower with tool access ($d = -0.31$ and $d = -0.42$ respectively), suggesting a potential positive effect that the study was underpowered to confirm with only 14 participants.

On the NASA-TLX results, a notable difference is the overall difference in the averages between the groups. Group 1 consistently responded with a higher workload than group 2, in both of the tasks, regardless of tool access. The reason for this is not necessarily clear. A post-hoc explanation for this could be that group 2 demographically indicated that they are more familiar with the notion of features and feature models than group 1. This can show why group 1 found task 1, where they had access to Agent HAnS as having a higher workload, as the notion of features added an extra dimension of complexity for them. Likewise group 2 then perceived the workload lower than group 1 for task 2 as they had a greater benefit of Agent HAnS as they were more familiar with the notion of features.

What stands to reason against this is that within the groups the perceived workload did not vary significantly between the tasks. This suggests that Agent HAnS had no effect on the perceived workload. The perceived workload is therefore most likely an inherent difference between the groups that wasn't measured in the user study. To mitigate this a larger number of respondents would be required to minimize the effect individual participants have on the overall results.

Next, for the SUS survey the result was an average of 73.9, which is above the threshold of 70 set in chapter 3. Because the standard deviation was 10.5, some of the participants (5 out of 14) responded with a SUS score below 70. Looking at the individual dimensions of the SUS survey shows that while the tool was easy to use, the statement *I think that I would like to use this system frequently* was the only statement to receive a below neutral average score of 2.8, as per section 5.2.1. A reason for this could be that feature models are an unfamiliar concept that feels more like a distraction during the LLM-based code generation process. It is also possible that it is harder to appreciate the potential benefits of using such a tool in the short time span of the user study.

An overall low workload indicated by the NASA-TLX survey and a SUS score above 70 indicate that Agent HAnS is usable. Agent HAnS being present while generating code does not have a significant impact on perceived workload.

Conclusion

6.3 Threats to Validity

External Validity. This thesis was only performed at one company, Centiro. The user study only made use of one programming language (JavaScript) with one framework (React). Only Claude Code and Anthropic's models were used for the user study. Together, these limitations reduce the generalizability of the study. As discussed, the study did not cover longer periods of time and the insights to the long-term effects of using Agent HAnS are therefore limited. The number of participants also poses a threat to validity, as with small group sizes each individual participant has a measurable effect on the final result. A within-subjects crossover design was used to try and mitigate this and utilize the participants to the fullest.

Internal Validity. Experimenting using LLMs is inherently unstructured and non-deterministic. Reproducibility is therefore lowered as shifting a single parameter slightly can have effects on the results.

Both the code comprehension questions and the correctness evaluation were graded by the authors without any method for inter-rater reliability. This could reduce the objectivity of the grading.

Another concern with the code comprehension questions is confirmation bias. While the answer was compared to the relevant code, the grading became a judgment call. Partially correct answers could therefore have been scored as correct or incorrect depending on whether the participant had access to the tool or not. This could be improved by a blind grading.

Since tasks during the user study were conducted in the same order for both groups, a learning-effect could affect the results for task 2, as participants get more comfortable with Claude Code.

After the survey, answers to familiarity questions indicated that participants in group 2 had a higher familiarity with feature models than group 1 overall. This was not identified before the study and were therefore not controlled for in the analysis.

Construct Validity. The primary construct validity concern is whether or not the chosen instruments capture what they are intended to measure.

The domain-specific Likert-questions were designed to cover both trust and comprehension, but the distinction between them is not always clear. Since the statements covered both, no individual result could be drawn for trust and comprehension separately from this instrument.

For the correctness evaluation, the baseline was created by the authors. Since our experience with feature models is limited, the baseline sometimes proved to be of worse quality than the generated feature model. This would in turn result in a worse score based on our grading scale.

Finally, all participants were colleagues of the authors and from the same company, Centiro Solutions AB, where the thesis was conducted. This may have inflated positive responses and task results as participants may have been inclined to give a positive view of the tool.

6.4 Future Work

From the discussion items above and through other discussions during the process of this thesis a number of suggestions for future work has been brought up.

6.4.1 Improvements to the Tool

Agent HAnS as described in this thesis is a passive tool. It offers no way for the user to interact with the source code or the feature model itself. During the user study and the testing of the final cycle more ideas for improvements were brought up.

One idea is to extend Agent HAnS to support feature model actions inside the GUI. For example, the user could after a prompt rearrange the feature model with the tool, and the LLM would make those changes to the feature model and the code behind the scenes.

While the vibe coding philosophy is to generate code and revise after the fact, it is also possible to move some of the design decisions in relation to the feature model before code generation by letting the LLM present the feature model to the developer and ask for confirmation or revisions before proceeding. This could be as simple as adding an instruction to the global instruction file reminding the LLM to confirm the feature model before proceeding.

6.4.2 Long-term Studies

As mentioned earlier in the discussion, comprehension of projects becomes more difficult as projects grow larger and developers have to remember information over time. Due to the nature of the thesis we did not have the option to hold any longer term experiments that show how Agent HAnS affects developers' understanding of LLM-generated code.

One hypothesis is that Agent HAnS would have a larger effect on developers' comprehension and trust for projects that are being worked on for weeks or months, with tens of thousands lines of code, compared to the hundreds of lines in the user study for this thesis.

6.4.3 Larger Scale Benchmark

One challenging aspect during the thesis was trying to measure how good LLMs are at creating feature models. The first benchmark was too simple leading to all language models completing all tasks perfectly. The intention was to find which models were better or worse at generating relevant feature models, but as all results were the same the differences could not be observed.

It was therefore realized that a more complex benchmark was required. Creating such a benchmark is not trivial, especially finding a way to evaluate and judge the results. The first challenge is to define a sufficiently complex dataset to evaluate. The initial intention was to make use of a pre-existing dataset with annotated source code and corresponding feature models. We were not able to find a dataset which matched the requirements. The planned dataset was of too low quality with incomplete feature models and missing annotations in the source code.

Next it was attempted to try and find datasets of LLM conversations from generating software projects. While it wasn't possible to get a baseline the thinking was that existing projects could be used to define new feature models, and exported prompt sequences could be used to test Agent HAnS. Unfortunately, the available exported LLM conversations were sporadic and were from the middle of conversations. We required conversations that started from the beginning of a project, otherwise creating a relevant feature model would be unreasonable.

In the end, we had to settle for creating the conversations and feature models ourselves. Using a manually defined expectation as the baseline introduces a limitation by itself as we as authors of this thesis have a limited experience with feature models. For future work more effort can be made making a better benchmark where more time is spent on defining the ground truth feature models together with domain experts. A more extensive grading scale allows for better judgment of the quality of the feature models. One addition could be to make the grading scale consist of several dimensions. For example, differentiating between the locations of features in the model and the presence of features.

6.4.4 Feature Model Enriches Context for the LLM

When discussing Agent HAnS with other developers a question that was asked is whether the feature model is there only for the human developer or if it is also for the LLM. We did not consider this at the start of the thesis, but there is a possibility that not only can the feature model and embedded feature annotations serve as a guide for the human, but also act as a self-created guiding structure for the LLM. The hypothesis is that the feature model gives a faster way for the LLM to build its context to what features are present and where they are located, which in turn can let it make more well-reasoned modifications and work faster in a codebase.

A study could be conducted where a series of prompts are executed in sequence to the same models, where one of the agents has access to Agent HAnS and the other

doesn't. The completion times and overall results can be compared to see if the LLM gets any benefit by keeping a feature model recorded while generating code. Note that the opposite could also be true as participants in the user study of this thesis noted that sometimes the code generation felt slower as the LLM also had to take steps making the feature model instead of just generating code.

7

Conclusion

LLM-based coding processes, like vibe coding, have become more common in recent years. As AI-generated software projects become large, comprehension becomes challenging. This in turn reduces the trust in code generated by AI. In the domain of feature-oriented software engineering feature models and embedded feature annotations are established concepts for getting an overview of the features present in a codebase and easing traceability of said features. Feature models however, require constant maintenance to reflect their associated codebase.

This thesis presents Agent HAnS, a tool that automatically generates and maintains feature models and embedded feature annotations in a LLM-based software development workflow. In addition Agent HAnS presents a graphical dashboard between prompts highlighting changes made to the source code from a feature-oriented perspective. To evaluate the usability and impact on comprehension and trust of Agent HAnS, a user study with 14 professional developers was conducted. On comprehension and trust, no statistically significant effect was found within the context of the user study. A more extensive user study may have the possibility of more clearly showing the potential benefits of using Agent HAnS, as exemplified by one participant: *"The functionality of the tool did not provide me with a lot of value compared to me just reading the code, in an IDE, or in Claude Code. Maybe for bigger more complex tasks, but for this, I didn't feel the added value."* Nevertheless, Agent HAnS was shown to be a usable tool that did not significantly affect the workload of LLM-based coding workflows. To evaluate the quality of the feature models generated by the tool a correctness evaluation was conducted. Overall, Agent HAnS varies in performance, sometimes generating feature models equivalent to a human-defined baseline, and at other times missing critical features requiring manual intervention.

We hope that this thesis can serve as a foundation for further research on the intersection of feature-oriented software development and LLM-based coding. Longer studies on larger and more complex projects are still required in order to understand the full potential of Agent HAnS. For further development of Agent HAnS, we suggest extending the dashboard with interactive feature model actions where the developer can review and adjust the feature model before starting the implementation. For further research, we suggest an investigation into whether feature models and embedded feature annotations can benefit the LLM by providing a richer context for future prompts.

Bibliography

[1] Mathieu Acher and Jabier Martinez. Generative ai for reengineering variants into software product lines: An experience report. In *Proceedings of the 27th ACM International Systems and Software Product Line Conference - Volume B*, SPLC '23, page 57–66, New York, NY, USA, 2023. Association for Computing Machinery.

[2] Ahmad Al Shihabi, Jan Sollmann, Johan Martinson, Wardah Mahmood, and Thorsten Berger. An ide plugin for clone management. In *28th ACM International Systems and Software Product Line Conference*, SPLC '24, page 42–45. ACM, Sept 2024.

[3] Anomaly. https://opencode.ai, 2026. Accessed: 2026-02-06.

[4] Anomaly. https://opencode.ai/docs/skills/, 2026. Accessed: 2026-05-21.

[5] Anthropic. https://claude.com/product/claude-code, 2026. Accessed: 2026-02-06.

[6] Anthropic. https://code.claude.com/docs/en/skills, 2026. Accessed: 2026-05-21.

[7] Anysphere. https://cursor.com/, 2026. Accessed: 2026-02-06.

[8] Aaron Bangor, Philip Kortum, and James Miller. Determining what individual sus scores mean: Adding an adjective rating scale. *Journal of usability studies*, 4(3):114–123, 2009.

[9] Thorsten Berger, Daniela Lettner, Julia Rubin, Paul Grünbacher, Adeline Silva, Martin Becker, Marsha Chechik, and Krzysztof Czarnecki. What is a feature? a qualitative study of features in industrial software product lines. In *Proceedings of the 19th International Conference on Software Product Line*, SPLC '15, page 16–25, New York, NY, USA, 2015. Association for Computing Machinery.

[10] Thorsten Berger, Wardah Mahmood, Ramzi Abu Zahra, Igor Vassilevski, Andreas Burger, Wenbin Ji, Michal Antkiewicz, and Krzysztof Czarnecki. Cost and benefit of tracing features with embedded annotations. *ACM Trans. Softw. Eng. Methodol.*, August 2025. Just Accepted.

[11] Thorsten Berger, Wardah Mahmood, Johan Martinson, and Jude Gyimah. Fm-pro feature modeling process, technical documentation. 2024.

[12] John Brooke. Sus: A quick and dirty usability scale. *Usability Eval. Ind.*, 189, 11 1995.

[13] Paul Clements and Linda Northrop. *Software product lines*, volume 1. Addison-Wesley Boston, 2002.

[14] Jacob Cohen. *Statistical power analysis for the behavioral sciences*. routledge, 2013.

[15] Bogdan Dit, Meghan Revelle, Malcom Gethers, and Denys Poshyvanyk. Feature location in source code: a taxonomy and survey. *Journal of Software: Evolution and Process*, 25(1):53–95, 2013.

[16] easelab. embeddedannotationdatasetmain. `https://bitbucket.org/easelab/embeddedannotationdatasetmain/src/master/`, 2021. Accessed: 2026-05-21.

[17] Ahmed Fawzy, Amjed Tahir, and Kelly Blincoe. Vibe coding in practice: Motivations, challenges, and a future outlook–a grey literature review. *arXiv preprint arXiv:2510.00328*, 2025.

[18] GitHub. https://github.com/features/copilot/cli/, 2026. Accessed: 2026-02-06.

[19] Sandra G Hart. Nasa-task load index (nasa-tlx); 20 years later. In *Proceedings of the human factors and ergonomics society annual meeting*, volume 50, pages 904–908. Sage publications Sage CA: Los Angeles, CA, 2006.

[20] Sandra G Hart and Lowell E Staveland. Development of nasa-tlx (task load index): Results of empirical and theoretical research. In *Advances in psychology*, volume 52, pages 139–183. Elsevier, 1988.

[21] Kevin Hermann, Johan Martinson, and Thorsten Berger. Visualizing feature-oriented software evolution. In *Proceedings of the 2025 29th ACM International Systems and Software Product Line Conference-Volume A*, pages 136–141, 2025.

[22] Wenbin Ji, Thorsten Berger, Michal Antkiewicz, and Krzysztof Czarnecki. Maintaining feature traceability with embedded annotations. In *Proceedings of the 19th International Conference on Software Product Line*, SPLC '15, page 61–70, New York, NY, USA, 2015. Association for Computing Machinery.

[23] Andrej Karpathy. *https://x.com/karpathy/status/1886192184808149383*. 2025. Accessed: 2026-05-21.

[24] Eric Knauss. Constructive master's thesis work in industry: Guidelines for applying design science research, 2021.

[25] Jacob Krüger, Wanzi Gu, Hui Shen, Mukelabai Mukelabai, Regina Hebig, and Thorsten Berger. Towards a better understanding of software features and their characteristics: A case study of marlin. In *Proceedings of the 12th International Workshop on Variability Modelling of Software-Intensive Systems*, VAMOS '18, page 105–112, New York, NY, USA, 2018. Association for Computing Machinery.

[26] Jacob Krüger, Thorsten Berger, and Thomas Leich. *Features and How to Find Them: A Survey on Manual Feature Location*. 2019.

[27] Lukas Linsbauer, E. Roberto Lopez-Herrejon, and Alexander Egyed. Recovering traceability between features and code in product variants. In *Proceedings of the 17th International Software Product Line Conference*, SPLC '13, page 131–140, New York, NY, USA, 2013. Association for Computing Machinery.

[28] Stephane Maes. The gotchas of ai coding and vibe coding. it's all about support and maintenance, 05 2025.

[29] Johan Martinson, Kevin Hermann, Riman Houbbi, David Stechow, and Thorsten Berger. Lightweight visualization of software features with hans-viz. In *Proceedings of the 29th ACM International Systems and Software Product Line Conference - Volume B*, SPLC-B '25, page 31–34. ACM, Sept 2025.

[30] Johan Martinson, Herman Jansson, Mukelabai Mukelabai, Thorsten Berger, Alexandre Bergel, and Truong Ho-Quang. Hans: Ide-based editing support for embedded feature annotations. In *Proceedings of the 25th ACM International Systems and Software Product Line Conference-Volume B*, pages 28–31, 2021.

[31] Johan Martinson, Wardah Mahmood, Jude Gyimah, and Thorsten Berger. Fmpro: A feature modeling process. *IEEE Transactions on Software Engineering*, 51(1):262–282, 2025.

[32] Jacqueline Mitchell and Yasser Shaaban. Position: Vibe coding needs vibe reasoning: Improving vibe coding with formal verification. LMPL '25, page 84–90, New York, NY, USA, 2025. Association for Computing Machinery.

[33] Model Context Protocol, a Series of LF Projects, LLC. Understanding mcp clients. `https://modelcontextprotocol.io/docs/learn/client-concepts`, 2025. Accessed: 2026-03-17.

[34] Model Context Protocol, a Series of LF Projects, LLC. Understanding mcp servers. `https://modelcontextprotocol.io/docs/learn/server-concepts`, 2025. Accessed: 2026-03-17.

[35] Model Context Protocol, a Series of LF Projects, LLC. What is the model context protocol (mcp)? `https://modelcontextprotocol.io/`, 2025. Accessed: 2026-03-17.

[36] Mukelabai Mukelabai, Kevin Hermann, Thorsten Berger, and Jan-Philipp Steghöfer. Featracer: Locating features through assisted traceability. *IEEE Transactions on Software Engineering*, 49(12):5060–5083, 2023.

[37] Leonardo Passos, Krzysztof Czarnecki, Sven Apel, Andrzej Wąsowski, Christian Kästner, and Jianmei Guo. Feature-oriented software evolution. In *Proceedings of the 7th International Workshop on Variability Modelling of Software-Intensive Systems*, pages 1–8, 2013.

[38] Denys Poshyvanyk, Yann-Gael Gueheneuc, Andrian Marcus, Giuliano Antoniol, and Vaclav Rajlich. Feature location using probabilistic ranking of methods based on execution scenarios and information retrieval. *IEEE Transactions on Software Engineering*, 33(6):420–432, 2007.

[39] Francisca Pérez, Jorge Echeverría, Raúl Lapeña, and Carlos Cetina. Comparing manual and automated feature location in conceptual models: A controlled experiment. *Information and Software Technology*, 125:106337, 2020.

[40] Lekshmi Murali Rani, Richard Berntsson Svensson, and Robert Feldt. Ai for requirements engineering: Industry adoption and practitioner perspectives, 2025.

[41] Partha Pratim Ray. A review on vibe coding: Fundamentals, state-of-the-art, challenges and future directions. *Authorea Preprints*, 2025.

[42] Julia Rubin and Marsha Chechik. *A Survey of Feature Location Techniques*, pages 29–58. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.

[43] Ranjan Sapkota, Konstantinos I. Roumeliotis, and Manoj Karkee. Vibe Coding vs. Agentic Coding: Fundamentals and Practical Implications of Agentic AI, May 2025. arXiv:2505.19443 version: 1.

[44] Tobias Schwarz, Wardah Mahmood, and Thorsten Berger. A common notation and tool support for embedded feature annotations. SPLC '20, page 5–8, New York, NY, USA, 2020. Association for Computing Machinery.

[45] Johannes Stümpfle, Devansh Atray, Nasser Jazdi, and Michael Weyrich. Large language model assisted transformation of software variants into a software product line. In *2025 IEEE/ACM 22nd International Conference on Software and Systems Reuse (ICSR)*, pages 12–20, 2025.

[46] Vijay Vaishnavi and W Kuechler. Design science research in information systems. *Association for Information Systems*, 01 2004.

[47] Jinshui Wang, Xin Peng, Zhenchang Xing, and Wenyun Zhao. How developers perform feature location tasks: a human-centric and process-oriented exploratory study. *Journal of Software: Evolution and Process*, 25(11):1193–1224, 2013.

[48] Yangtian Zi, Luisa Li, Arjun Guha, Carolyn Anderson, and Molly Q Feldman. "i would have written my code differently': Beginners struggle to understand llm-generated code. In *Proceedings of the 33rd ACM International Conference on the Foundations of Software Engineering*, FSE Companion '25, page 1479–1488, New York, NY, USA, 2025. Association for Computing Machinery.

A

Interview Questions

Background:
Can we anonymously use the answers of this interview for our research?
How long have you been a professional software developer?

Core:
Do you know what vibe coding is?
Have you tried vibe coding?
If no:
-Why not?
If yes:
-How often do you vibe code?

Which tools and which models do you use?
What tasks do you use vibe coding for and what do you avoid using vibe coding for?
Have you experienced any issues or difficulties when vibe coding?
What would make vibe coding more attractive to you?

As part of our Master Thesis we are going to build a tool that non-intrusively creates a feature model and embedded feature annotations while you vibe code. The idea is that the feature model and annotations will help structure the project to allow for better maintainability and traceability of features.

When developing this tool, what would you like us to keep in mind?
What features would you appreciate in such a tool?
Is there someone else you think we should interview, that has an interesting or valuable perspective to these questions?

B

LLM benchmark

B.1 Discussion

Ideally, more tests would have been run trialing more models, but two issues blocked progress. For one, running the benchmark was surprisingly slow in relation to the difficulty of the task. This is likely due to using large models which need to reason about a task even if it is simple. A typical task took 10-30 seconds. All together this means running the full 90 task test suite took around half an hour. The second issue is that the LLM provider used (Github Copilot) measures its request quota not on number of generated tokens or time spent executing a prompt but on number of prompts sent. Therefore making many small individual requests drained the quota quickly even if the execution time for the 90 prompts can be compared to that of a single large and complex prompt which would have only costed one request.

B.2 Results

All models completed all tasks successfully, as seen in Table B.1. This likely due to the tasks being simple. Even though Claude Sonnet 4.5 shows some scores below 100%, the failing tasks showed on closer inspection that the feature model task itself was completed successfully. The reason the benchmarking program deemed the task a failure was because the model made changes to some additional files, for example a readme-file with the added functionality of the new feature described. These additions were not technically incorrect, but as they weren't part of the benchmark this lead to the tests failing since the benchmark fails the test if the model decides to change any files it's not supposed to. The conclusion we drew from this, as well as from our interviews and personal observations is that the Claude models have a tendency to work beyond the instructions, and therefore require stricter instructions limiting what they should do.

The overall conclusion we drew was that model selection based on feature modelling capability does not seem consequential, at least for the simple tasks described above. Running a larger scale benchmark would likely give deeper insights but was deemed out of scope for this thesis as more complex tasks increases the number of possibilities of correct solutions to check for exponentially. Additionally, defining

Task	GPT 5.2	Gemini 2.5 Pro	Claude Sonnet 4.5
Add feature	100%	100%	80%
Remove feature	100%	100%	100%
Rename feature	100%	100%	90%

Table B.1: Task completion rate from the benchmark of the language models on feature model tasks

what an embedded feature annotation being correct means is not clear either. As mentioned in the background, there are many ways to define what a feature is and how to divide a software project into features.

C

Identified challenges

The following is a list of the challenges identified during the problem understanding phase of the thesis. The list is based on challenges identified through literature, our own experiences and interviews with developers at Centiro. A total of seven interviews were conducted with developers that had different opinions and amounts of experience with vibe coding. Most of these challenges were identified during the first cycle, but developed understanding of the problem in the later cycles led to a few revisions and additions.

C.1 Limitations from providers

A number of the identified challenges come from limitations made by the organizations responsible for the infrastructure in which LLMs live in and are difficult to address by the user, with the exception of spending more money. Over time as LLMs become more efficient and cheaper these barriers will become lower.

Limited context space
LLMs are inherently limited to how much data they can keep in their context at one time.

Running out of tokens
All LLM providers limit the amount of tokens or requests a user can send during a given time frame, which could be measured in hours or months. Some of our interviewees found this to become a challenge when working purely with vibe coding.

Slow generation
Executing on a prompt can take anywhere from a few seconds to several tens of minutes. The models can become faster over but a current challenge is that generating code can be time consuming, for more complex prompts and tasks.

C.2 How models act

Some of the challenges identified come from the inherent way that LLMs work that lead them to making errors that a human intelligence typically wouldn't make.

Many of these challenges occur depending on how the models are prompted and can therefore be mitigated from different prompting strategies.

Hallucinations
Hallucinations are when an LLM generates seemingly true information which has no ground in reality. In the software development domain this can mean generating code which makes use of imported packages or functions that don't exist or aren't implemented.

Getting stuck in infinite loops
We found that the LLMs can at times get stuck in infinite loops where they make a step-by-step plan and then right before they start to execute the steps, the model reminds itself that it should plan before executing. This can occur again and again leading to an endless loop of no code being generated.

Getting distracted
Some models more than others have a tendency of being overly ambitious and completing more tasks than what they were asked. This can often be avoided through more explicit prompts, still, it is a challenge. An example here is that we found during the benchmark that Anthropic's models often edited the readme files when being asked to add embedded feature annotations.

Forgetting instructions
During long sessions without clearing the context window the LLM had a tendency to forget instructions that it was given early in the conversation. We noticed this through experimentation when creating large projects, the LLM often forgot to update the feature model and the annotations after prompting for a while.

Tunnel vision
The opposite of getting sidetracked is being too narrow and LLMs can do this as well and not noticing that a specific feature exists in several places and needs to be addressed at all of them, and not just the perceived main one.

Getting code production ready
While LLMs are good at getting fully featured functional applications up and running, Fawzy et al. [17] found that the models had a hard time finishing the final steps needed to ensure production readiness.

C.3 User familiarity, understanding and trust

Many of the challenges mentioned during the interviews and in literature center around a lack of understanding the generated code and the lack of trust which follows from it.

Understanding generated code
Vibe coding can quickly generate a large amount of code, which can lead to difficulties for the developer to understand the entire project. Especially for less experienced

developers.

Lack of trust
From the more skeptical interviews a common challenge was the perceived lack of trust. Interviewees felt that they could not be certain that the LLM followed their instructions as they intended, especially due to the nature of LLMs being non-deterministic leading to different results for the prompt furthers the feeling of uncertainty.

QA practices Fawzy et al. [17] found that QA practices were more easily missed for projects which makes use of AI generated code.

Writing descriptive prompts
Writing prompts can be both challenging and time consuming, especially if you're not experienced. The prompts need to cover the right words to make sure the LLM acts as expected.

Not understanding current workflow state
When defining a workflow through a prompt or with the use of instructional files, it can be hard to follow along where the model is in that workflow. This can make it harder to understand the LLMs process.

D

Survey

Introduction

The following is the user study survey form and instruction for group 2. The form is mostly the same for group 1 except that the tool is used in task 1 instead of task 2.

Master thesis AI coding experiment

First of all, thank you for participating in our study!

We are Jacob and Moltas and we're on our final year of the Software Engineering master program at Chalmers University of Technology. Our master thesis is about ways to structure LLM-based coding techniques using features. This experiment is a part of the final evaluation of our thesis.

The overall structure looks like this:

- Setup & consent
- Task 1, and a few questions
- Task 2, and a few questions
- Questions about the experiment as a whole & background information

In each task you will implement some changes in an existing codebase via vibe coding (Claude Code) and be asked about your comprehension of said codebase. For one of the tasks you will have access to a tool we have developed during this thesis. In the other task you won't, as that acts as a control.

The total time to perform this experiment is around 1–2 hours, where the bulk of the time will (hopefully) be spent on the two tasks.

Good luck!

Q1. Consent *(Required)*

Do you consent to participating in this study, and having your results used anonymously as part of our Master Thesis?

☐ Yes

Q2. Name *(Required)*

While you will be completely anonymous in our report, we need to know your name to connect your generated code from this experiment to your answers in this survey.

(Free text)

Setup and Installation

Before you get started with the actual tasks, we need to ensure you are properly set up. In this experiment you will mainly use Claude Code. Use Claude Sonnet 4.6 as your model (LLM).

Please disable any MCP-servers and plugins you have for this experiment, as they might interfere. Pure quality of life additions, like Azure DevOps and LSP-extensions are okay.

Download the repository

For the experiment you need some files, which we provide in the form of a git repository. Clone the two task repositories to a suitable location on your computer. You will during the study submit the code that you generate to a branch with your name.

Q3. I have downloaded the repositories *(Required)*

☐ Yes

Task 1

Open the repo for task 1 and create a new branch using your name or username to ensure it's unique. After each upcoming step in this experiment, make a commit to the branch you just created, named after the completed step.

You will use Claude Code to extend a base application based on requirements/instructions in the following steps. All code changes must be made by Claude Code, meaning no manual coding.

Before you move on, run the application using `npm install` and then `npm run dev` and explore for a maximum of 5 minutes what the application for task 1 looks like.

Task 1 — Step 1

Above the card list add a filtering system to enable filtering of the displayed recipes based on meal type: Breakfast / Lunch / Dinner. Add the capability of having multiple shopping lists. Add a button on each recipe card to import all its ingredients directly into a shopping list with the name of the recipe. At the top right of the page, add a search bar to find specific recipes.

You are done with this step when all capabilities described above are implemented and in working order; verify by running the application.

After each step, there will be one question. Do not ask Claude Code to answer these questions, as we are looking for your human judgement. Limit yourself to 5 minutes for these comprehension questions.

Q4. I have created a commit with my work for this step *(Required)*
Commit message: `step 1`

☐ Yes

Q5. If you needed to add a "Brunch" filter option, what would need to change? *(Required)*

(Free text)

Task 1 — Step 2

Extend the filtering system to also support dietary preferences: e.g. vegetarian, vegan, gluten-free. Secondly, merge the search function and the filtering system into the same panel.

You are done with this step when all capabilities described above are implemented and in working order; verify by running the application.

After each step, there will be one question. Do not ask Claude Code to answer these questions, as we are looking for your human judgement. Limit yourself to 5 minutes for these comprehension questions.

Q6. I have created a commit with my work for this step *(Required)*
Commit message: `step 2`

☐ Yes

Q7. Did the LLM put search and filter in a single component or keep them in separate files rendered side by side? *(Required)*

(Free text)

Task 1 — Step 3

Separate the drinks recipes from the meal recipes into separate views. Recipes for food have a cook time and meal type, whereas for cocktails there should be a description for glass type and ABV content.

You are done with this step when all capabilities described above are implemented and in working order; verify by running the application.

After each step, there will be one question. Do not ask Claude Code to answer these questions, as we are looking for your human judgement. Limit yourself to 5 minutes for these comprehension questions.

Q8. I have created a commit with my work for this step *(Required)*
Commit message: `step 3`

☐ Yes

Q9. If you needed to add a "Mocktail" category for non-alcoholic drinks, what would need to change? *(Required)*

(Free text)

Task 1 — Step 4

Merge the multiple shopping lists back into a single shopping list, which updates when importing from recipes, while the option for manual entry remains. It should also support grouping items in the list by location in a grocery store.

You are done with this step when all capabilities described above are implemented and in working order; verify by running the application.

After each step, there will be one question. Do not ask Claude Code to answer these questions, as we are looking for your human judgement. Limit yourself to 5 minutes for these comprehension questions.

Q10. If a recipe's ingredient list is edited after its ingredients were already imported, does the shopping list reflect the change? *(Required)*

(Free text)

Q11. I have created a commit with my work for this step *(Required)*
Commit message: `step 4`

(Marks end of Task 1. Commit and push your branch before continuing.)

Questions for Task 1 (without tool access)

NASA-TLX

Q12. NASA-TLX *(Required)*

Rate each dimension on a scale from 0 to 100, where 0 means very low and 100 means very high.

Dimension	0–100
How mentally demanding was the task?	
How physically demanding was the task?	
How hurried or rushed was the pace of the task?	
How successful were you in accomplishing what you were asked to do? *(0 = failure, 100 = perfect)*	
How hard did you have to work to accomplish your level of performance?	
How insecure, discouraged, irritated, stressed and annoyed were you?	

Domain Questions

Q13. Domain questions *(Required)*

Rate each statement from 1 (Strongly disagree) to 5 (Strongly agree).

Statement	1	2	3	4	5
I have a good understanding of what the generated code looks like and how it is structured.					
I could have explained the changed code to another developer.					
I felt uncertain about what was happening in the codebase at some point during the task.					
I trust that all parts of the codebase relevant to my request were addressed.					
I knew where I would look if I wanted to verify the changes that were made.					
I would feel confident handing this codebase off to another developer.					

Feature Model Introduction

A short lesson on feature models

A *feature* is an abstract way to describe a capability of a software.

A *feature model* is a named tree structure that organises the capabilities or features of a software system into a shared conceptual hierarchy. Each node in the tree is a feature — a meaningful unit of functionality at some level of granularity. The tree itself has no special semantics beyond containment: a parent feature is simply a logical grouping of its children.

For a web server, the tree might look like:

```
WebServer
├── Auth
│   ├── Basic
│   └── OAuth
└── Cache
    ├── Redis
    └── Memory
```

This structure gives an indication of how a piece of software is structured and how the different parts relate to one another. Rather than relying on file structure or naming conventions to convey what a module does and where it fits in the system, the feature model makes that explicit and uniform. The tree structure is typically stored in a `.feature-model` file in the root of a project.

Embedded feature annotations connect that tree back to source code via structured comments that tag code with its corresponding feature:

```
// &begin[OAuth]
public string ExchangeToken(string code) { ... }
// &end[OAuth]

// &begin[Redis]
public class RedisClient { ... }
// &end[Redis]

// &begin[Auth]
public void Authenticate(HttpRequest request) { ... }
// &end[Auth]
```

This gives traceability in both directions: given a feature, you can find all code that implements it; given a piece of code, you immediately know where it sits in the system's conceptual map. Because the annotations are embedded directly in the source code, they evolve with the code through version control, code review, and any downstream tooling — including LLMs, which can use them as grounding context when generating or modifying code.

How the tool works

When the tool is activated, at the beginning of a response it will reply "Using the feature model workflow" to show that it is working. It will then apply some feature

modelling practices as it fulfils your prompt.

If it has changed or generated new code, it will at the end of the response activate a web server and take you to a webpage showing the feature model and affected files. Try clicking around — it's interactive.

If this webpage never shows up when you are doing the upcoming task, something is wrong. Contact us.

Activating the Tool

Before starting task 2, make sure the tool is activated. It should be installed inside the relevant repository by default.

To check if it is working: open the task 2 repository in Claude Code, then type `/mcp` and see if the "fm-gui" MCP shows up in the list. If so, ensure it is activated. If not, contact us.

Reminder: if you have other MCPs activated, please disable them during the experiment, as they might interfere.

Q14. I can see "fm-gui" in my MCP list *(Required)*

☐ Yes

Q15. I think I have a good enough understanding of feature models to continue *(Required)*
If it still feels unclear, contact us.

☐ Agree

Task 2

Before starting with task 2, take a 5-minute break to stretch your legs.

Open the repo for task 2 and create a new branch using your name or username to ensure it's unique. You will use Claude Code to extend a base application based on requirements/instructions in the following steps. All code changes must be made by Claude Code, meaning no manual coding.

Before you move on, run the application using `npm install` and then `npm run dev` and explore what the application for task 2 looks like.

Task 2 — Step 1

Add a priority system to the cards with levels: High, Medium, and Low. Each priority level should be colour coded so users can quickly distinguish urgency at a

glance. Add the capability of setting a due date on each card. Add a button on each card to quickly cycle through priority levels.

You are done with this step when all capabilities described above are implemented and in working order; verify by running the application.

After each step, there will be one question. Do not ask Claude Code to answer these questions, as we are looking for your human judgement. Limit yourself to 5 minutes for these comprehension questions.

Q16. I have created a commit with my work for this step *(Required)*
Commit message: `step 1`

☐ Yes

Q17. If you needed to add a fourth priority level, what would need to change? *(Required)*

(Free text)

Task 2 — Step 2

Extend the card details to replace due dates with deadlines. Cards whose deadlines have passed should be clearly marked with a red overdue label so users can immediately identify at-risk tasks.

You are done with this step when all capabilities described above are implemented and in working order; verify by running the application.

After each step, there will be one question. Do not ask Claude Code to answer these questions, as we are looking for your human judgement. Limit yourself to 5 minutes for these comprehension questions.

Q18. Were any existing files modified when replacing due dates with deadlines, or was the change purely additive? *(Required)*

(Free text)

Q19. I have created a commit with my work for this step *(Required)*
Commit message: `step 2`

☐ Yes

Task 2 — Step 3

Separate the summary view from the full detail view. Cards on the board should only display the title and ID, whereas a modal opened by clicking a card should

contain all of the card's data.

You are done with this step when all capabilities described above are implemented and in working order; verify by running the application.

After each step, there will be one question. Do not ask Claude Code to answer these questions, as we are looking for your human judgement. Limit yourself to 5 minutes for these comprehension questions.

Q20. Is the modal a separate component or inline conditional rendering? *(Required)*

(Free text)

Q21. I have created a commit with my work for this step *(Required)*
Commit message: `step 3`

☐ Yes

Task 2 — Step 4

Merge priority and deadline into a single unified urgency indicator, calculated from both values, with the stages Critical, High, and Normal. The urgency indicator should be displayed on the cards in the list.

Remove the ability for cards to be deleted, since removing them would erase the history of a project.

You are done with this step when all capabilities described above are implemented and in working order; verify by running the application.

After each step, there will be one question. Do not ask Claude Code to answer these questions, as we are looking for your human judgement. Limit yourself to 5 minutes for these comprehension questions.

Q22. Did the LLM merge the existing implementations for deadline and priority or create an entirely new solution for urgency? *(Required)*

(Free text)

Q23. I have created a commit with my work for this step *(Required)*
Commit message: `step 4`

(Marks end of Task 2. Commit and push your branch before continuing.)

Questions for Task 2 (with tool access)

NASA-TLX

Q24. NASA-TLX *(Required)*

Rate each dimension on a scale from 0 to 100, where 0 means very low and 100 means very high.

Dimension	0–100
How mentally demanding was the task?	
How physically demanding was the task?	
How hurried or rushed was the pace of the task?	
How successful were you in accomplishing what you were asked to do? *(0 = failure, 100 = perfect)*	
How hard did you have to work to accomplish your level of performance?	
How insecure, discouraged, irritated, stressed and annoyed were you?	

Domain Questions

Q25. Domain questions *(Required)*

Rate each statement from 1 (Strongly disagree) to 5 (Strongly agree).

Statement	1	2	3	4	5
I have a good understanding of what the generated code looks like and how it is structured.					
I could have explained the changed code to another developer.					
I felt uncertain about what was happening in the codebase at some point during the task.					
I trust that all parts of the codebase relevant to my request were addressed.					
I knew where I would look if I wanted to verify the changes that were made.					
I would feel confident handing this codebase off to another developer.					

System Usability Scale

Q26. System Usability Scale *(Required)*

Rate each statement from 1 (Strongly disagree) to 5 (Strongly agree).

By "system" we refer to the tool and the workflow of being shown the feature model after every prompt.

Statement	1	2	3	4	5
I think that I would like to use this system frequently.					
I found the system unnecessarily complex.					
I thought the system was easy to use.					
I think that I would need the support of a technical person to be able to use this system.					
I found the various functions in this system were well integrated.					
I thought there was too much inconsistency in this system.					
I would imagine that most people would learn to use this system very quickly.					
I found the system very cumbersome to use.					
I felt very confident using the system.					
I needed to learn a lot of things before I could get going with this system.					

Questions about the Feature Model

Q27. Questions about the feature model *(Required)*

Rate each statement from 1 (Strongly disagree) to 5 (Strongly agree), or select "I don't know".

Statement	1	2	3	4	5	I don't know
The features shown in the feature model felt relevant and of quality.						
There was code that belonged to a feature that was not annotated.						

Q28. Any comments to provide nuance to the above questions?

(Free text)

Q29. What did you like about using the tool?

(Free text)

Q30. Did you have any challenges using the tool? If yes, please describe.

(Free text)

Q31. Any suggestions to improve the tool in the future?

(Free text)

Final Questions and Background Information

Now you have completed both tasks. We have some final questions about your background before you submit the survey.

Q32. How many years have you worked professionally as a developer? *(Required)*

- <1
- 1–2
- 3–5
- 6–10
- 10+

Q33. How familiar were you with generating code through prompting LLMs, prior to this user study? *(Required)*

We are thinking of workflows similar to "vibe coding", where the majority of code being written inside a project is created by an LLM which you prompt — not just having code completions enabled in your IDE, but towards the end of using Claude Code or strong use of the chat window in your IDE.

Scale: 1 (Unfamiliar) — 2 — 3 — 4 — 5 (Familiar)

Q34. How familiar were you with the following, prior to this user study? *(Required)*

	1	2	3	4	5
The notion of features					
Feature models					
Embedded feature annotations					

Scale: 1 = Unfamiliar, 5 = Familiar