OxidOWL: Development and Maintenance of OWL DL Reasoners with Coding Agents

Riccardo Sieve[1]

[1]University of Oslo, Gaustadalléen 23B, Oslo, 0373, Norway

Abstract

Description Logic reasoners enable automated reasoning over ontologies expressed using the Web Ontology Language (OWL). They can be used to infer logical consequences from a set of asserted facts or axioms within an ontology. However, many well known reasoners are currently non-maintained, or are not actively developed. This can lead to unpatched bugs, missing features, or non-adherence to possible new standards that might arise in the future. In this paper, we analyse and show how we can employ coding agents for the development and maintenance of OWL DL reasoners. To this end, we propose OxidOWL, an OWL DL reasoner developed and maintained through coding agents. We validate our approach through thorough testing and compare the proposed approach with existing well-known reasoners.

Keywords

Description Logic, OWL DL Reasoner, Coding Agents

1. Introduction

OWL DL reasoners, given their ability to infer logical consequences from a set of asserted facts or axioms [1], enable the inference of knowledge of an application domain in a structured way [2]. Pellet [3], Hermit [4, 5], FaCT++ [6], and Konclude [7] with their support with different DLs like $\mathcal{ALC}$ [8], $\mathcal{SHOIN}$ [9], and $\mathcal{SROIQ}$ [10], are widely used for their efficiency and robustness in handling complex ontologies.

However, many DL reasoners are currently unmaintained or abandoned [11]. Lack of maintenance can result in compatibility issues with newer versions of OWL, challenges in addressing bugs [12, 13, 14] and optimisations [15], and can affect software in general [16]. In this regard, the increase in the development of coding agents and AI-assisted programming tools can help mitigate some of these challenges by automating parts of the development and maintenance process.

In this paper, we present OxidOWL, an OWL DL reasoner developed and maintained through coding agents that aims to address these challenges. We evaluate our reasoner with thorough testing against well-known frameworks, highlighting both its effectiveness and efficiency in handling various reasoning tasks, as well as its shortcomings.

To this aim, we consider the following research questions:

RQ1: How can coding agents be effectively utilised to develop and maintain OWL DL reasoners and check that these reasoners are correct? (*design and implementation aspects*)

RQ2: How does the performance of the AI-assisted reasoner compare to traditional reasoners in terms of reasoning tasks? (*benchmarking and evaluation*)

RQ3: What are the limitations and challenges of using coding agents for OWL DL reasoner development and maintenance? (*development feasibility*)

The main contributions of this paper are threefold. First, we present an OWL DL reasoner developed and maintained using coding agents, demonstrating the potential of AI-assisted programming in this

DL 2026: 39th International Workshop on Description Logics, July 17–19, 2026, Lisbon, Portugal

riccasi@ifi.uio.no (R. Sieve)

https://sievericcardo.github.io/ (R. Sieve)

0009-0000-8683-1902 (R. Sieve)

domain. Second, we provide a comprehensive evaluation of our reasoner, highlighting its strengths, weaknesses, and areas for improvement. Third, we discuss the usage of coding agents for the development of OWL DL reasoners, providing insights into the challenges and opportunities presented by introducing coding agents in the software development pipeline.

The rest of the paper is organised as follows: Section 2 provides the necessary background on DL, reasoners, and coding agents. Section 3 highlights the methodology used for the development and maintenance with coding agents. Section 4 describes the design and implementation of our reasoner. Section 5 presents the validation process and the results obtained. Finally, Section 6 discusses the lessons learnt during the process, and Section 7 concludes the paper with future work directions.

2. Background and Related Work

DL is a decidable, expressive subset of First-Order Logic (FOL) [17] that provides formal knowledge representation languages. Its expressiveness has been already studied and analysed [18, 19], and it has been shown that DL can be used to represent the numerical constraints. Concepts in DL are represented in terms of axiom types, which can be categorised into TBox and ABox axioms [2].

OWL DL Reasoners. OWL DL reasoners support tasks such as consistency checking, classification, and instantiation [20]. Over time, several OWL DL reasoners have been developed, including Pellet [3], HermiT [4], FaCT++ [6], and Konclude [7], which have been studied for their efficiency and robustness in handling various reasoning tasks [21, 22]. Additionally, approaches such as justification-based techniques [23, 24] and delta-reasoning [25] have been widely used to enhance the performance of reasoning tasks. OWL reasoners have also been integrated with other systems, such as rule engines [26], and to support more complex reasoning scenarios and domain-specific languages [27, 28]. While several profiles have been developed to reduce the complexity of OWL DL reasoners [29, 30, 31, 32], many reasoners are currently not actively developed or abandoned, leading to unpatched bugs, or a lack of emerging features like RDF-star[1] and RDF 1.2[2].

Coding Agents. To facilitate the development and maintenance of software systems, coding agents and AI-assisted programming tools can be employed to automate parts of the development process. Coding agents, such as GitHub Copilot [33, 34] and Claude [35, 36], leverage large language models (LLMs) to generate code snippets, suggest improvements, and assist with debugging. These tools can help developers by providing code suggestions based on natural language descriptions, reducing the time and effort required for coding tasks. LLMs and coding agents have proven effective for testing [37, 38, 39], repair [40, 41], and development [42, 43]. Because they are context-aware and have access to the whole software codebase, software agents can analyse existing code and provide relevant suggestions, which makes them capable of handling more complex programming tasks for code generation. This has opened up new possibilities for automating various aspects of software development. Tools such as RALPH[3] and KISS AI[4] can be used to manage the development process, track changes, and ensure that the codebase remains consistent and up-to-date. By leveraging coding agents, developers can focus on higher-level design and logic while automating routine coding tasks, leading to faster development cycles and improved software quality. Coding agents can also assist in maintain scientific software, which is often complex and requires ongoing maintenance to ensure its reliability and relevance [44].

In this paper, we want to fill the gap between coding agents and DL reasoner development and maintenance by presenting OxidOWL, a DL reasoner developed and maintained through coding agents for reasoning tasks over OWL ontologies. This work aims to highlight the potential of coding agents

[1] https://w3c.github.io/rdf-star/cg-spec/editors_draft.html
[2] https://www.w3.org/TR/rdf12-concepts/
[3] https://github.com/snarktank/ralph
[4] https://github.com/ksenxx/kiss_ai/

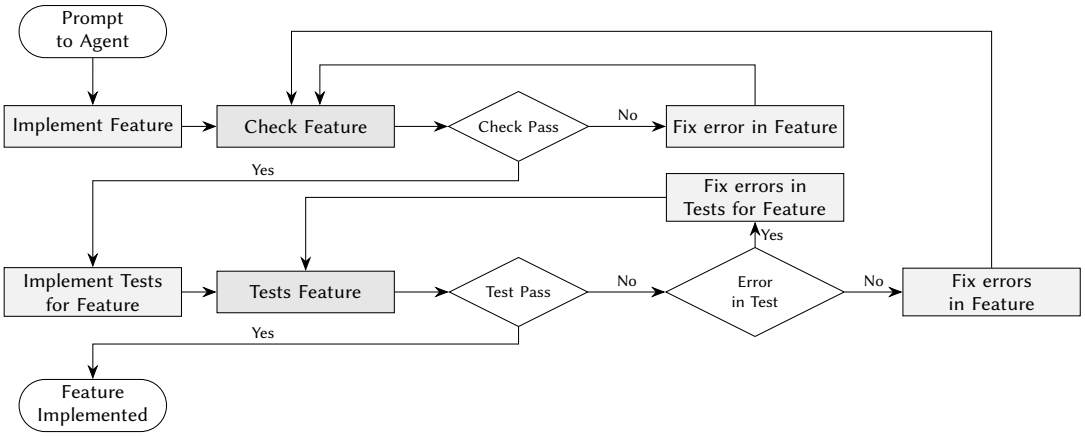


Figure 1: Execution flow of the development process for a feature or implementation. The prompt can be structured as: *Add support for insert and delete queries in sparql.* It can further be structured as *Implement the cache for the reasoner using LRU eviction policy while ensuring that it supports incremental reasoning to improve the performance of the reasoner.* Overall, for more complex tasks, a more detailed prompt would lead to better outcomes.

in the development of complex software systems, and to provide insights into how these tools can be effectively used in the context of ontology engineering and web semantics.

3. Methodology

In this section, we detail the development and maintenance process of software with coding agents. For both development and maintenance processes, we used GitHub Copilot Pro, leveraging Claude Sonnet as agent. The development started with Claude Sonnet 3.7. During the overall process, the agent improved, and Claude Sonnet 4 and Claude Sonnet 4.5 were later used. For the implementation, we used Rust as programming language due to its performance, safety features, and strong support for concurrent programming, which is beneficial for building a robust reasoner. For the current development, we only iterated through a single agent, requiring manual intervention to check the output and provide feedback. Furthermore, the developers performed thorough testing, and the output was used to provide feedback to the agent.

3.1. Software Development with Coding Agents

The development process, shown in Fig. 1, follows an iterative process in which different features are first developed by the agent, and then checked to ensure that no bugs were introduced. If bugs are found, the agent deals with them by inspecting the command line output. After this check passes, tests are created and run. They follow the same pattern to ensure that both the feature and the tests correctly solve the request.

Through several iterations, we used coding agents to generate the codebase, implement algorithms, and perform testing and debugging. This approach allowed for rapid prototyping and iterative refinement of the reasoner's components. However, coding agents might not fully understand the request, leading to errors in the logic of the program. For example, if the prompt was not precise enough, the resulting implemented code could result in unexpected behaviours. Overall, the development process involved the following steps:

- **Requirement Analysis** to define the functionalities and features required for the reasoner. This process involved specifying the supported DL constructs, reasoning tasks, and performance goals.
- **Code Generation** to generate code and modules based on the defined requirements.
- **Testing and Debugging** done by employing coding agents to create test cases, identify bugs, and suggest fixes.

- **Optimisation** to refine the code for performance improvements and implement additional features as needed.
- **Documentation** to facilitate future maintenance and development.

This iterative approach enabled us to build a functional and efficient DL reasoner while leveraging the capabilities of coding agents to enhance the development workflow. Furthermore, since coding agents organise the development process in a structured way, it allows for better tracking of progress and easier identification of areas that require further attention or improvement. In this sense, coding agents proved to be effective in building a complex software system like OxidOWL.

3.2. Software Maintenance with Coding Agents

We used coding agents not only for the initial development of OxidOWL but also for its ongoing maintenance. Coding agents assist in identifying bugs and suggesting code improvements, as well as in implementing new features. This collaborative approach can help ensure that the reasoner remains up-to-date with the latest advancements in DL reasoning and continues to meet user needs effectively. Figure 2 shows the process that the coding agent follows during the maintenance request. For the maintenance process, we followed an iterative approach similar to that in the development phase:

- **Issue Identification**, where we can either use coding agents to help identify bugs, performance bottlenecks, and areas for improvement in the codebase, or users can report issues directly in the chat. Here, GitHub Copilot with Claude Sonnet proved to be effective tools. Through shell integration, they can analyse execution and build output to identify and correct issues.
- **Code Review** by employing coding agents to review the existing code to suggest optimisations and ensure adherence to coding standards.
- **Feature Implementation** by providing requirements for features to extend the existing codebase, and using coding agents to generate the necessary code.
- **Testing and Validation** to ensure that changes do not introduce new issues and that the reasoner continues to function correctly.
- **Documentation Updates** to reflect any changes made to the codebase, ensuring that users and developers have access to accurate information.

Overall, the use of coding agents in both the development and maintenance of OxidOWL enhanced the development and maintenance of the reasoner, allowing for rapid iteration and continuous improvement. The total time spanned approximately six months. This proved to be a viable approach for building complex software systems like OWL DL reasoners, demonstrating the potential of coding agents in software engineering tasks. To ensure the correctness of the reasoner output, testing solutions were employed. More information is provided in Sect. 5.

3.3. Automating the Planning Phase

To refine the development process and to generate a structured plan, the `plan` feature integrated within GitHub Copilot with Claude Sonnet helps in creating a more structured prompt. The planning process involved the following steps: (1) definition of requirements and objectives; (2) generation of a comprehensive development plan outlining the key milestones, tasks, and timelines; (3) in each iteration, further questions, generally three, are posed to the user on how to better refine the prompt and take decisions for the desired outcome; (4) based on the feedback, the agents revise the plan to ensure that it aligns with the project goals; (5) finally, the finalised development plan is generated, and used to guide the implementation process.

The planning process can significantly enhance the development process by providing a clear roadmap and ensuring that all aspects of the project are considered from the outset. Furthermore, since the document produced is already structured as a markdown file, it can be directly integrated into the

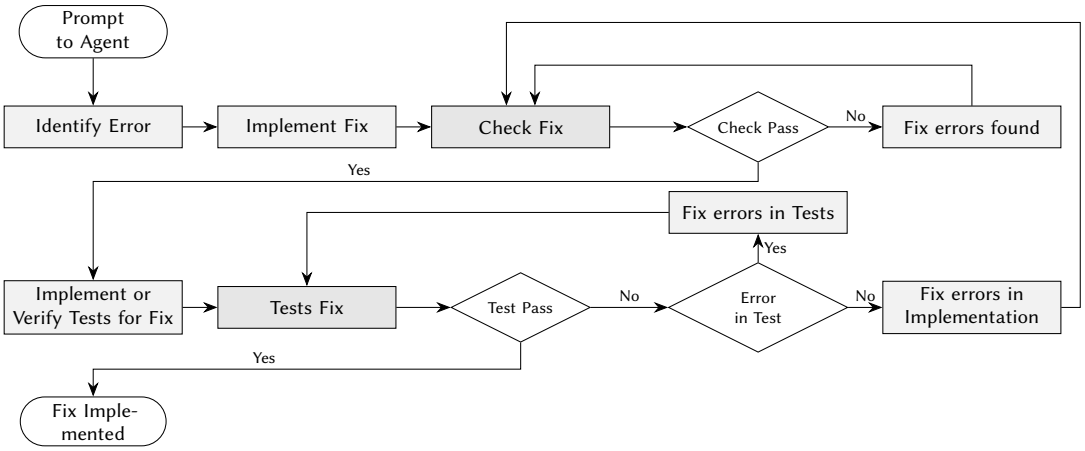


Figure 2: Execution flow of the maintenance process for a feature or implementation. The prompt can be structured as: *Executing the command* `oxidowl -input ...` *generates the following error output .....* The prompt can be further refined with *Identify the cause and fix the error*, but generally providing the error message is enough to proceed to the fixing phase. For improvements, prompts can be structured as *Make a deep analysis for the project. Analyse when* `unwrap()` `expect()` *or* `clone()` *have been used. Check if they can be replaced for better efficiency and safety. Make a thorough analysis of the project to identify possible elements that might be modified to improve the overall efficiency and safety of the project.*

project, facilitating easy access and reference. This approach not only streamlines the planning phase but also ensures a prompt that is best suited for coding agents to follow.

During the development of OxidOWL, we used the planning phase only for later maintenance as, at the time of development, the feature was not yet present and planning was done by prompting the agent to plan the roadmap through files.

4. OxidOWL

In this section, we describe the design and implementation of OxidOWL[5]. We outline the architecture, key components, and usage of the reasoner for different tasks. The resulting software of approximately 106,976 lines of code (excluding tests) is completely done by the coding agents, where the code is fully generated by the coding agents and the portion of code written by the developer is negligible (i.e. less than 1%). To develop the reasoner, we used the material as input described in Sect. B.

4.1. Architecture Overview

OxidOWL is a tableau-based reasoner for the Description Logic $\mathcal{SROIQV}$ built to work with OWL 2 ontologies following the hyper-tableau algorithm [45] employed in HermiT [46], while incorporating optimisations from other reasoners like Konclude [7]. The high-level architecture of OxidOWL, depicted in Fig. 3, consists of the following key components:

- **Public API Layer:** This layer provides interfaces for users to interact with the reasoner, including ontology loading, query processing, and reasoning task execution.
- **Reasoner Core:** The core component responsible for managing reasoning tasks, coordinating between different services, and handling communication with the API layer.
- **Saturation Engine**: This module performs pre-processing of ontologies to apply deterministic rules and optimisations before invoking the tableau algorithms.
- **Tableau Algorithm:** This module implements the hyper-tableau algorithm and other reasoning strategies, providing the necessary logic for inference.

[5]The code for the reasoner can be found on GitHub at https://github.com/sievericcardo/oxidowl

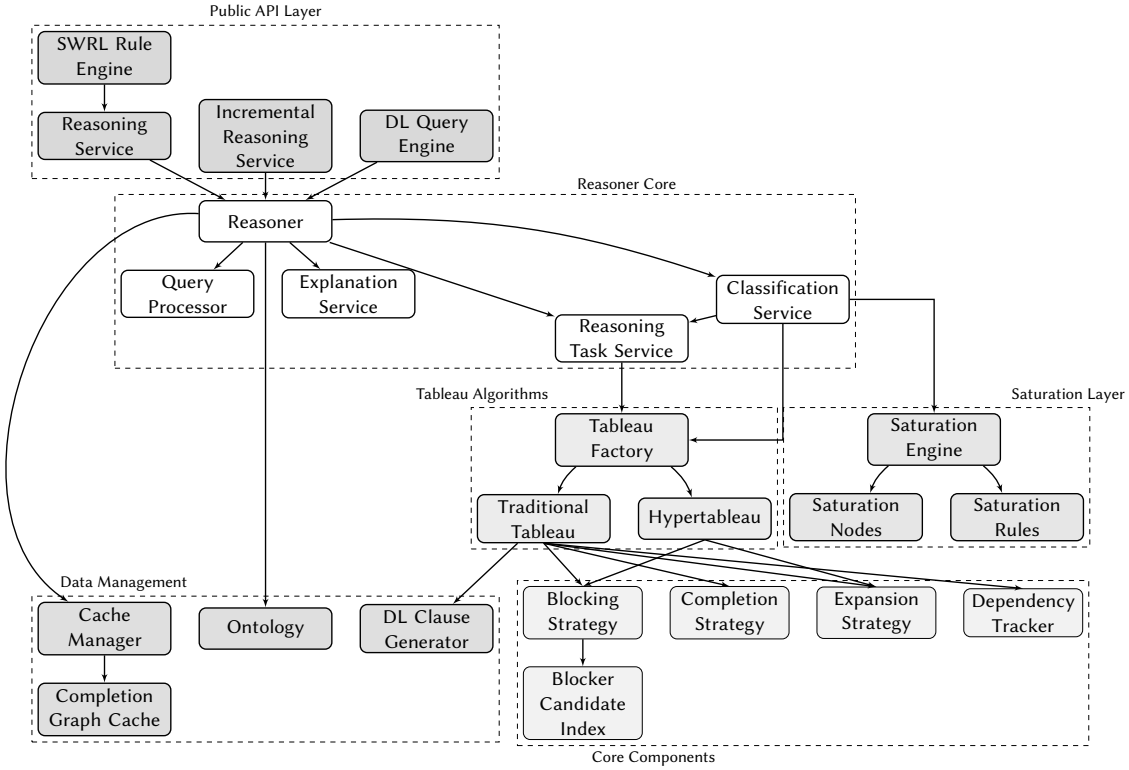


Figure 3: High-Level Architecture of the OxidOWL Reasoner. The diagram illustrates the main components and their interactions within the reasoner. Each layer is encapsulated in a dashed box and each sub-components are highlighted in different shades for clarity and clear separation.

- **Core Components:** These components include strategies for blocking, completion, expansion, and dependency tracking, which are essential for the tableau reasoning process.
- **Data Management:** This module handles ontology storage, caching, and DL clause generation, ensuring efficient data access and manipulation during reasoning.

Finally, we integrated in the reasoner a dedicated triplestore that manages data and facilitates query processing, which can be enabled or disabled based on user requirements. This allows SPARQL queries to interact with ontologies, enabling users to perform complex queries and retrieve inferred knowledge efficiently.

The overall development and maintenance of OxidOWL, shown in Tbl. 1, highlights the overall quality of the generated codebase. Specifically, elements like the Tableau Algorithm, were not fully developed at first by the agent, leaving many functions or features as TODO. This was caused by both a limited context space (smaller than 150K token), while the agent had to keep track of the whole application that had to be developed. In this regard, a planning phase, or a more granular approach can help mitigating it, as the different steps gets treated as single tasks, allowing for an optimisation of context compression.

4.2. Using the Reasoner

OxidOWL is designed to be usable both as a standalone application through a Command Line Interface (CLI) and as a library integrated into other software systems. Users can interact with the reasoner through its Public API Layer, which provides functions to load ontologies, execute reasoning tasks, and retrieve results. The standalone application allows users to perform reasoning tasks directly from the command line, supporting different tasks such as consistency checking, classification, following the tasks available in HermiT.

Table 1
Analysis of the quality of the generated code of Claude Sonnet within GitHub Copilot during the implementation of the different main components of OxidOWL. The table defines the capability of the agent on implementing with a scale that goes from 1 to 5, where 1 means that the agent was not able to implement the component properly and required a several refinements from the developer through several requests, 3 means that the agent was able to implement the component albeit with some errors and missing features, and 5 means that the agent was able to implement the component correctly. The last column describes the interactions done by the developer to ensure a correct implementation.

Component	Agent Capability	Human Interaction
Public API Layer	5	No further interaction from the Human was required for the Public API Layer.
Reasoner Core	3	While the first code was easily computed, several errors were given for not fully adhering to Rust patterns.
Saturation Engine	5	While the Saturation Engine was brought up at a later time during the optimisation phase of the reasoner, Claude Sonnet was able to correctly implement it.
Tableau Algorithm	2	The first version of the algorithm was lacking. Most functions were kept as TODOs, many errors in Rust were present, and some aspects of the reasoning algorithms were not fully implemented. Major interactions were required to make the implementation complete.
Core Components	4	While the agent did not cover all element and all cases, the generated code was working correctly.
Data Management	5	The Data Management Layer was well implemented by the agent. Some components have been modified over time to improve the performance, but almost no supervision was required.

The library allows developers to integrate OxidOWL into their applications, allowing users to leverage the reasoning capabilities of OxidOWL in various contexts, from web applications to data analysis tools. Through the configuration file, the user can define the type of reasoning algorithm, blocking strategy, and expansion strategy. It further allows to use incremental reasoning, define the maximum expansion depth, and caching options. Moreover, the user can enable or disable the triplestore integration based on their needs and define the port number for SPARQL query access.

Currently, OxidOWL supports OWL 2 ontologies serialised in RDF/XML, Turtle, and OWL Functional Syntax formats. Future plans include expanding support to additional serialisation formats and enhancing the reasoner's capabilities to handle larger and more complex ontologies efficiently.

5. Validation

In this section, we describe how we answered the research questions, and the corresponding results obtained. Finally, we consider the current limitations that can arise in the process.

5.1. Experiment Setup

For the development and maintenance of the reasoner we used the add-on of GitHub Copilot for Visual Studio Code, allowing us to have the complete set of agents to employ directly in the development environment. For performance evaluation, we setup OxidOWL to be used as a standalone CLI application, generating the binary with `cargo build --release` to ensure that an optimised binary was generated. We evaluated the system on a Linux Server with 8 cores and 16 GB or RAM.

Table 2
Results of the language-based testing (a) and mutation-based testing (b) conducted on OxidOWL to verify its capacity to parse ontologies and correctly handles mutated ontologies. For mutation-based testing, we employed HermiT and Konclude as oracles.

Format	Tests run	Pass Rate
Functional Syntax	22	27%
OWL/XML	18	39%
Turtle	24	35%
RDF/XML	20	35%
CrossSyntax	10	50%

(a) Results of the first language-based testing campaign on OxidOWL. The second campaign had a rate between 75 and 100%. We then achieved 100% in every category in the third campaign.

Metric	Value
Parse errors	801
Annotation errors	∼500
SWRL errors	∼220
IRI errors	∼60
Equiv-disj bugs	∼1600

(b) Findings of the mutation-based testing campaigns on OxidOWL over more than 50,000 tests.

5.2. Results

To answer **RQ1**, we tested the reasoner to verify both the quality of the produced result and to highlight possible errors or areas of improvement to further refine and maintain the reasoner. To this end, we tested OxidOWL in a two-fold manner: (1) we tested the reasoner with a set of ontologies to ensure the correctness of the reasoner with respect to OWL grammar; and (2) we evaluated the reasoner to correctly handle reasoning tasks with unexpected inputs.

We applied language-based testing [12] to generate a set of ontologies that would capture both valid and invalid constructs of the OWL 2 DL grammar. We generated a set of ontologies that would cover the whole grammar, and used them as input for OxidOWL. The reasoner was initially not able to handle all the constructs. Through several iterations and three campaigns, we were able to achieve full coverage of the OWL generated grammar, demonstrating the effectiveness of coding agents in developing and maintaining the reasoner.

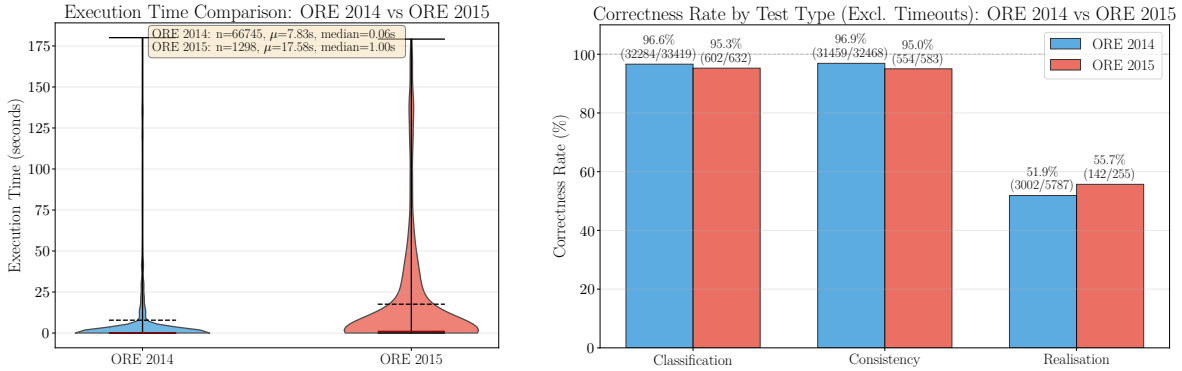
To evaluate OxidOWL against different types of input, we further applied mutation-based testing, a testing technique that creates new tests cases by changing part of existing ones, generating mutations [13]. We applied this technique to generate a set of ontologies by applying mutations. These mutations included adding, removing, or modifying axioms in ontologies. We then used these mutated ontologies as input for OxidOWL to evaluate its robustness and ability to handle unexpected inputs.

Within the first campaign, many mutation patterns were initially not handled correctly by OxidOWL. Following the testing results, we improved the reasoner to fix the issues through several iterations. We proceeded with seven testing campaigns until we were able to handle all mutation patterns, further demonstrating the effectiveness of coding agents in maintaining the reasoner. The overall result of both language-based testing and mutation-based testing are shown in Tbl. 2, where we detail the type of errors identified during the tests. Within both testing campaigns, we were able to identify several errors in the reasoner, including parsing errors, annotation errors, SWRL errors, and IRI errors. Some of these errors were due to the fact that we implemented custom parsers for some of the OWL syntaxes, which were not fully available within the libraries used. However, this proved helpful in showing how we could integrate them with coding agents, and how they can be effectively used to identify and address these issues.

While both language-based and mutation-based testing were helpful in identifying bugs in the grammar and in the language, they might not be covering OWL as a whole. As such, we plan to integrate these tests alongside the OWL conformance recommendation.[6]

To answer **RQ2**, we evaluated the performance of OxidOWL (version 0.10.0) against the OWL Reasoner Evaluation (ORE) 2014 [47] and the OWL Reasoner Evaluation (ORE) 2015 [48] benchmark suites. The benchmark suites consist of a set of ontologies and reasoning tasks, including *classification*, *consistency checking*, and *instance retrieval* (identified as Realisation). We conducted two testing cam-

[6] https://www.w3.org/TR/owl2-conformance/.



(a) Execution time distribution of the OxidOWL reasoner on the ORE 2014 and 2015 competition dataset. The dashed line represents the mean execution time.

(b) Correctness rate of the OxidOWL reasoner on the ORE 2014 and 2015 competition dataset. To check if the tests were correct, we excluded the ones that resulted in timeout.

Figure 4: Benchmark of OxidOWL with the ORE 2014 and ORE 2015 dataset. (a) captures the execution time of OxidOWL within the different dataset, and (b) highlights, for each category, the percentage of correct tests. We limit the execution time to 180 seconds, resulting in a failure if the reasoner took more than that time.

paigns of tests with two sets of timeouts to ensure that all tests would end in a timely manner and to verify the entire coverage of tests from OxidOWL, namely 180 seconds and two hours.

The results of the first campaign are shown in Fig. 4. The results showed good execution time for the task OxidOWL was able to solve, albeit with a high number of timeouts (overall, the completion rate for ORE 2014 was 83.4%, and 51.0% for ORE 2015[7]). Overall, while the results indicate that OxidOWL performs well, they also highlight areas for improvement in terms of handling a wider range of reasoning tasks and improving scalability, especially for realisation tasks.

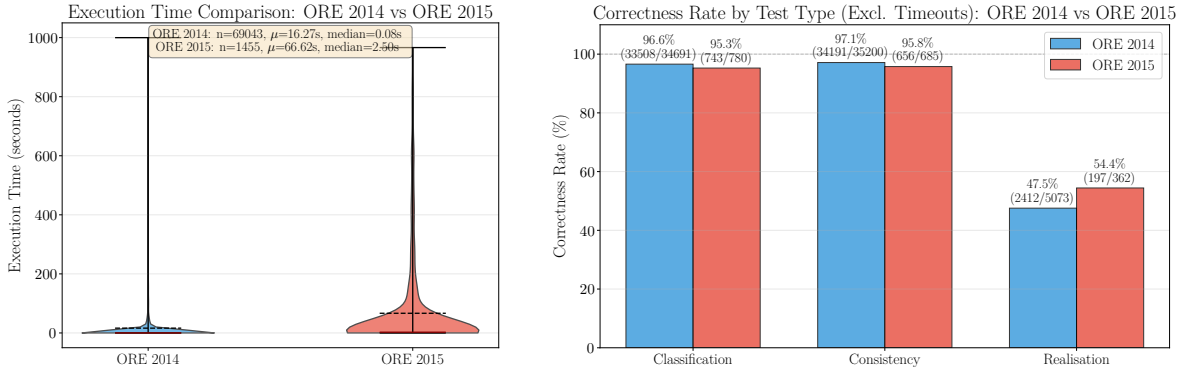
The main set of errors that we encountered during the execution include:

- Errors in parsing some prefix starting with "^^" (e.g. ^^xsd) that resulted in failure to parse the input ontology;
- Unmatched parentheses when line breaking, resulting in improper parsing of the ontology;
- Errors with *relative IRI* that lead to an unrecognised element; and
- Stack overflow during the realisation task due to missing depth checking and lack of efficient implementation.

The results of the second campaign with two hours for the timeout are shown in Fig. 5. With a longer timeout, OxidOWL was effective in completing a greater number of tasks, leading to a completion rate of 91.0% for ORE 2014 (albeit on a subset of the whole dataset) and 62.7% for ORE 2015. However, the results also indicate that OxidOWL still struggles to complete some tasks in a reasonable time frame, especially for ORE 2015. This suggests that while OxidOWL has viable performance, there are still areas for improvement in terms of handling more complex reasoning tasks and improving scalability, especially for realisation tasks. The results show that OxidOWL is a promising reasoner, but further work is needed to enhance its performance and scalability for a wider range of reasoning tasks.

The results for the second campaign might appear counter-intuitive due to the lower number of correct results compared to the 180 seconds campaign within the realisation tasks. This is mainly due to the fact that some of the aforementioned errors occurred within the longer timeout, causing a lower number of correct tests. Nevertheless, the errors identified during both campaigns are still valuable, as we can use the data to patch them, improve areas that were highlighted during the testing part, and ensure a more thorough implementation using the two datasets as a test platform.

[7]While the experiments were conducted under different conditions, the completion rate for ORE 2015 was 78% for HermiT and 96% for Konclude



(a) Execution time distribution of the OxidOWL reasoner on the ORE 2014 and 2015 competition dataset. The dashed line represents the mean execution time.

(b) Correctness rate of the OxidOWL reasoner on the ORE 2014 and 2015 competition dataset. The results show a higher correctness rate in every task type.

Figure 5: Benchmark of OxidOWL with the ORE 2014 and ORE 2015 dataset. (a) captures the execution time of OxidOWL within the different dataset, and (b) highlights, for each category, the percentage of correct tests. We limit the execution time to 2 hours, resulting in a failure if the reasoner took more than that time.

To answer **RQ3**, we analysed the development process of OxidOWL using coding agents. We inspected the code generated by the coding agents, both in terms of correctness and efficiency. For the former, we used unit tests, for the latter, by verifying the code against Rust best practices. We also evaluated the time taken to develop and maintain the reasoner using coding agents. We chose to build a reasoner without having thorough knowledge of how DL reasoners are normally implemented. This allowed us to fully harness coding agents without imposing our knowledge or way of working during the development and maintenance process. Time-wise, the overall development spanned a 6 months period, none of which was full time for the development of the reasoner.

On the correctness of the code generated by coding agents, while coding agents were able to generate code that was syntactically correct, there were instances where the generated code did not fully adhere to the intended logic or design patterns. This required human intervention to instruct the coding agents accordingly to enforce certain design principles or to correct logical errors.

Furthermore, we observed that coding agents sometimes produced code that could result in potential errors. Specifically to Rust, we noticed that the generated code was using instances of `unwrap()` that could lead to runtime panic if not handled properly[8].This required a thorough review of the generated code to ensure robustness and safety, which are critical aspects in Rust programming.

Finally, while coding agents were able to generate code quickly, they usually write fewer and larger files, leading to a more monolithic codebase, which could be hard to maintain in the long term. This required additional effort to refactor the codebase to improve maintainability.

5.3. Threats to Validity

While our evaluation demonstrates the potential of coding agents in developing and maintaining a DL reasoner, there are several aspects to be considered:

Internal Threats. The development process using coding agents may require a certain level of expertise and familiarity with the tools. This could limit the development for people who do not have experience with coding agents or the specific programming language used. Furthermore, the evaluation process was able to identify several errors in the reasoner. This was possible thanks to the extensive material available to do so. However, when developing a reasoner or scientific software with less

[8] https://web.archive.org/web/20260119221027/https://blog.cloudflare.com/18-november-2025-outage/

documentation and resources available, it might be harder to identify and address such issues, which could impact the overall quality of the output software developed with coding agents.

External Threats. The effectiveness of coding agents may vary depending on the specific tools and models used. Different coding agents may have different capabilities and limitations, which could impact the development process and the quality of the generated code. Furthermore, the ontologies used in the evaluation may not fully represent the diversity of real-world ontologies, which could limit how well OxidOWL performs in practical applications.

6. Discussion

In this section, we highlight the lessons learnt from developing and maintaining an OWL reasoner with coding agents.

Prompt quality. The quality of the prompt used to instruct the coding agents had a significant impact not only on the quality of the generated code but also on the speed of development. We identified, for well-formed input, that the amount of detail in the requirements, as well as the context provided to the coding agents, would improve the quality of the generated code. In this regard, well-formed prompts that included a clear description of the functionality to implement, the expected input and output, and any constraints or design patterns to follow. This emphasises the importance of prompt engineering in leveraging coding agents effectively.
Insight: To use coding agents to develop and maintain a DL reasoner, the prompt provided to the coding agents should be well-formed, with clear and detailed requirements regarding the different OWL constructs to support, the reasoning tasks to implement, and the expected performance goals.

Context. The quality of the input prompt has been found crucial to achieve good results from coding agents. Providing additional material, such as code snippets, documentation, or examples, proved to further enhance the quality of the generated code. During the development of OxidOWL, we provided additional documentation related to OWL, RDF, RDFS, SWRL, and DLs to the coding agents (for a complete list, see Sect. B). This allowed the coding agent to refer to the official specifications when generating the code, ensuring compliance with standards and best practices. Furthermore, we also identified relevant crates[9] to use during implementation with the corresponding documentation, which the coding agents could leverage to implement specific functionalities more effectively. We provided the crate firsthand, since coding agents might not have the full extent of the currently developed libraries for the chosen programming language.
Insight: Context is crucial for coding agents to generate high-quality code. Providing additional material as shown in Sect. B can significantly enhance the quality of the generated code by allowing coding agents to refer to relevant documentation and examples.

Further refinement. Coding agents were able to generate code quickly. However, the generated initial code often required further refinement and optimisation to meet the desired quality standards. Among these, review of the code for correctness, efficiency, and adherence to best practices. Coding agents might not always follow the intended design patterns or architectural principles, which requires human intervention to guide them in aligning the code with the overall design goals and best practices.
Insight: While coding agents can generate code quickly, the generated code often requires further refinement and optimisation to meet the desired quality standards. This includes reviewing the code for correctness, efficiency, and adherence to best practices, as well as iteratively refining the code to improve its structure and modularity. To address this, at least partially, it is possible to provide coding agents with specific design patterns and architectural principles to follow, which can help guide the code generation process and reduce the need for extensive refinement.

[9] A crate is a package of Rust code that can be imported as a library.

Human-in-the-loop. Despite advancements in coding agents, human oversight and intervention remained crucial throughout the development process. Human developers played a key role in guiding the coding agents, reviewing the generated code, and ensuring that it met the desired quality standards. However, by means of sub-agents, coding agents can be used to automate the testing process, and to provide feedback to the main agent by making them more autonomous. This can further reduce the need for human intervention and allow for a more efficient development process. In this regard, we plan to explore the use of multi-agent systems in future work.
Insight: Human oversight and intervention remain crucial when using coding agents for software development. Human developers can guide the coding agents, review the generated code, and ensure that all components are appropriately integrated and meet the desired quality standards.

Applicability to specialised tasks. Coding agents demonstrated the ability to handle specialised tasks, such as developing a DL reasoner, which requires domain-specific knowledge and expertise. However, the effectiveness of coding agents in specialised domains depended on the availability of relevant data and knowledge. In this regard, developing and maintaining a specialised reasoner with agents like GitHub Copilot and Claude Sonnet could be feasible, provided that the coding agents have access to sufficient domain-specific information and context.
Insight: Coding agents can be effective in developing and maintaining scientific software. Their effectiveness depends on the availability of relevant data and domain knowledge. In the context of developing a DL reasoner, this reflects to the availability of documentation, code snippets, and examples related to OWL, RDF, RDFS, SWRL, and DLs, as well as relevant libraries and tools.

7. Conclusion

In this paper, we presented OxidOWL, an OWL DL reasoner developed and maintained using coding agents. We discussed the design and implementation of the reasoner, highlighting the role of coding agents in the development process. Our evaluation demonstrated that OxidOWL performs comparably to traditional DL reasoners in terms of reasoning tasks, showcasing the potential of coding agents for both development and maintenance in this domain.

We further showed how coding agents can help developers in the development and maintenance of complex software systems where the domain knowledge is crucial and the software is often complex and requires ongoing maintenance to ensure its reliability and relevance. We also identified current limitations and challenges associated with using coding agents for development.

Moreover, this paper only covered the development and maintenance of an OWL DL reasoner, and the test of its features. In future, we plan on integrating OxidOWL with existing semantic web frameworks and tools, to facilitate its adoption in real-world applications, and to explore the potential of coding agents in enhancing interoperability and usability in broader areas.

Furthermore, while OxidOWL was able to perform reasoning tasks in a timely manner for the completed tasks, we plan to investigate its scalability in handling larger and more complex ontologies, as well as its performance in real-time reasoning scenarios. This will help us understand the limitations and potential optimisations for coding agents in the context of reasoning systems. We also plan on testing OxidOWL with a wider range of reasoners, to further evaluate its performance and identify areas for improvement.

Finally, we aim to explore the applicability of coding agents in developing reasoners for other logic-based formalisms, such as rule-based systems or temporal logics, to further assess their versatility and effectiveness in various domains of knowledge representation and reasoning.

Declaration on Generative AI

GitHub Copilot and Claude Sonnet were used for development and maintenance of OxidOWL and for code completion and rapid prototyping. No Generative AI was used to write the content of the paper.

References

[1] J. Z. Pan, I. Horrocks, Reasoning in the SHOQ(D_n) description logic, in: Proceedings of the 2002 International Workshop on Description Logics (DL2002), Toulouse, France, April 19-21, 2002, volume 53 of *CEUR Workshop Proceedings*, CEUR-WS.org, 2002.

[2] F. Baader, I. Horrocks, U. Sattler, Description logics, in: Handbook of Knowledge Representation, volume 3 of *Foundations of Artificial Intelligence*, Elsevier, 2008, pp. 135–179. doi:10.1016/S1574-6526(07)03003-9.

[3] E. Sirin, B. Parsia, Pellet: An OWL DL reasoner, in: Proceedings of the 2004 International Workshop on Description Logics (DL2004), Whistler, British Columbia, Canada, June 6-8, 2004, volume 104 of *CEUR Workshop Proceedings*, CEUR-WS.org, 2004.

[4] R. D. C. Shearer, B. Motik, I. Horrocks, HermiT: A highly-efficient OWL reasoner, in: Proceedings of the Fifth OWLED Workshop on OWL: Experiences and Directions, collocated with the 7th International Semantic Web Conference (ISWC-2008), Karlsruhe, Germany, October 26-27, 2008, volume 432 of *CEUR Workshop Proceedings*, CEUR-WS.org, 2008.

[5] I. Horrocks, B. Motik, Z. Wang, The HermiT OWL reasoner, in: Proceedings of the 1st International Workshop on OWL Reasoner Evaluation (ORE-2012), Manchester, UK, July 1st, 2012, volume 858 of *CEUR Workshop Proceedings*, CEUR-WS.org, 2012.

[6] D. Tsarkov, I. Horrocks, Fact++ description logic reasoner: System description, in: Automated Reasoning, Third International Joint Conference, IJCAR 2006, Seattle, WA, USA, August 17-20, 2006, Proceedings, volume 4130 of *Lecture Notes in Computer Science*, Springer, 2006, pp. 292–297. doi:10.1007/11814771_26.

[7] A. Steigmiller, T. Liebig, B. Glimm, Konclude: System description, J. Web Semant. 27-28 (2014) 78–85. doi:10.1016/J.WEBSEM.2014.06.003.

[8] K. Wu, V. Haarslev, A parallel reasoner for the description logic ALC, in: Proceedings of the 2012 International Workshop on Description Logics, DL-2012, Rome, Italy, June 7-10, 2012, volume 846 of *CEUR Workshop Proceedings*, CEUR-WS.org, 2012.

[9] H. Wang, Z. Li, A semantic matchmaking method of web services based on SHOIN$^+$ (D), in: Proceedings of The 1st IEEE Asia-Pacific Services Computing Conference, APSCC 2006, December 12-15, 2006, Guangzhou, China, IEEE Computer Society, 2006, pp. 26–33. doi:10.1109/APSCC.2006.17.

[10] D. T. Cucala, B. C. Grau, I. Horrocks, Sequoia: A consequence based reasoner for SROIQ, in: Proceedings of the 32nd International Workshop on Description Logics, Oslo, Norway, June 18-21, 2019, volume 2373 of *CEUR Workshop Proceedings*, CEUR-WS.org, 2019.

[11] K. Abicht, OWL reasoners still useable in 2023, CoRR abs/2309.06888 (2023). doi:10.48550/ARXIV.2309.06888. arXiv:2309.06888.

[12] T. John, E. B. Johnsen, E. Kamburjan, D. Steinhöfel, Language-based testing for knowledge graphs, in: The Semantic Web - 22nd European Semantic Web Conference, ESWC 2025, Portoroz, Slovenia, June 1-5, 2025, Proceedings, Part II, volume 15719 of *Lecture Notes in Computer Science*, Springer, 2025, pp. 24–46. doi:10.1007/978-3-031-94578-6_2.

[13] T. John, E. B. Johnsen, E. Kamburjan, RDFMutate : Mutation-based generation of knowledge graphs, in: The Semantic Web - ISWC 2025 - 24th International Semantic Web Conference, Nara, Japan, November 2-6, 2025, Proceedings, Part II, volume 16141 of *Lecture Notes in Computer Science*, Springer, 2025, pp. 295–312. doi:10.1007/978-3-032-09530-5_17.

[14] T. John, E. B. Johnsen, E. Kamburjan, Mutation-based testing of knowledge graphs, Empir. Softw. Eng. 31 (2026). doi:10.1007/s10664-026-10810-w.

[15] A. M. Ileri, H. McGinty, VEL: A formally verified reasoner for EL++ description logic, in: Proceedings of the ISWC 2024 Posters, Demos and Industry Tracks: From Novel Ideas to Industrial Practice co-located with 23nd International Semantic Web Conference (ISWC 2024), Hanover, Maryland, USA, November 11-15, 2024, volume 3828 of *CEUR Workshop Proceedings*, CEUR-WS.org, 2024. URL: https://ceur-ws.org/Vol-3828/paper22.pdf.

[16] S. Demeyer, C. D. Roover, M. Beyazit, J. Härtel, Threats to instrument validity within "in silico"

research: Software engineering to the rescue, in: Leveraging Applications of Formal Methods, Verification and Validation. Software Engineering Methodologies - 12th International Symposium, ISoLA 2024, Crete, Greece, October 27-31, 2024, Proceedings, Part IV, volume 15222 of *Lecture Notes in Computer Science*, Springer, 2024, pp. 82–96. doi:10.1007/978-3-031-75387-9_6.

[17] F. Baader, F. D. Bortoli, The expressive power of description logics with numerical constraints over restricted classes of models, in: Frontiers of Combining Systems - 15th International Symposium, FroCoS 2025, Reykjavik, Iceland, September 29 - October 1, 2025, Proceedings, volume 15979 of *Lecture Notes in Computer Science*, Springer, 2025, pp. 22–39. doi:10.1007/978-3-032-04167-8_2.

[18] F. Baader, A formal definition for the expressive power of terminological knowledge representation languages, J. Log. Comput. 6 (1996) 33–54. doi:10.1093/LOGCOM/6.1.33.

[19] F. Baader, A formal definition for the expressive power of knowledge representation languages, in: 9th European Conference on Artificial Intelligence, ECAI 1990, Stockholm, Sweden, 1990, 1990, pp. 53–58.

[20] D. Tsarkov, I. Horrocks, DL reasoner vs. first-order prover, in: Proceedings of the 2003 International Workshop on Description Logics (DL2003), Rome, Italy September 5-7, 2003, volume 81 of *CEUR Workshop Proceedings*, CEUR-WS.org, 2003.

[21] E. Jiménez-Ruiz, B. C. Grau, I. Horrocks, On the feasibility of using OWL 2 DL reasoners for ontology matching problems, in: Proceedings of the 1st International Workshop on OWL Reasoner Evaluation (ORE-2012), Manchester, UK, July 1st, 2012, volume 858 of *CEUR Workshop Proceedings*, CEUR-WS.org, 2012.

[22] G. Stoilos, B. C. Grau, B. Motik, I. Horrocks, Repairing ontologies for incomplete reasoners, in: The Semantic Web - ISWC 2011 - 10th International Semantic Web Conference, Bonn, Germany, October 23-27, 2011, Proceedings, Part I, volume 7031 of *Lecture Notes in Computer Science*, Springer, 2011, pp. 681–696. doi:10.1007/978-3-642-25073-6_43.

[23] M. Lee, N. Matentzoglu, B. Parsia, U. Sattler, A multi-reasoner, justification-based approach to reasoner correctness, in: The Semantic Web - ISWC 2015 - 14th International Semantic Web Conference, Bethlehem, PA, USA, October 11-15, 2015, Proceedings, Part II, volume 9367 of *Lecture Notes in Computer Science*, Springer, 2015, pp. 393–408. doi:10.1007/978-3-319-25010-6_26.

[24] M. Lee, N. Matentzoglu, U. Sattler, B. Parsia, Verifying reasoner correctness - A justication based method, in: Informal Proceedings of the 4th International Workshop on OWL Reasoner Evaluation (ORE-2015) co-located with the 28th International Workshop on Description Logics (DL 2015), Athens, Greece, June 6, 2015, volume 1387 of *CEUR Workshop Proceedings*, CEUR-WS.org, 2015, pp. 46–52.

[25] B. Motik, I. Horrocks, S. M. Kim, Delta-reasoner: a semantic web reasoner for an intelligent mobile platform, in: Proceedings of the 21st World Wide Web Conference, WWW 2012, Lyon, France, April 16-20, 2012 (Companion Volume), ACM, 2012, pp. 63–72. doi:10.1145/2187980.2187988.

[26] G. Meditskos, N. Bassiliades, Combining a DL reasoner and a rule engine for improving entailment-based OWL reasoning, in: The Semantic Web - ISWC 2008, 7th International Semantic Web Conference, ISWC 2008, Karlsruhe, Germany, October 26-30, 2008. Proceedings, volume 5318 of *Lecture Notes in Computer Science*, Springer, 2008, pp. 277–292. doi:10.1007/978-3-540-88564-1_18.

[27] E. Kamburjan, A. Pferscher, R. Schlatte, R. Sieve, S. L. Tapia Tarifa, E. B. Johnsen, Semantic reflection and digital twins: A comprehensive overview, in: The Combined Power of Research, Education, and Dissemination: Essays Dedicated to Tiziana Margaria on the Occasion of Her 60th Birthday, volume 15240 of *Lecture Notes in Computer Science*, Springer, 2025, pp. 129–145. doi:10.1007/978-3-031-73887-6_11.

[28] R. Sieve, E. Kamburjan, F. Damiani, E. B. Johnsen, Declarative dynamic object reclassification, in: 39th European Conference on Object-Oriented Programming, ECOOP 2025, June 30 to July 2, 2025, Bergen, Norway, volume 333 of *LIPIcs*, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2025, pp. 29:1–29:31. doi:10.4230/LIPICS.ECOOP.2025.29.

[29] J. Mendez, B. Suntisrivaraporn, Reintroducing CEL as an OWL 2 EL reasoner, in: Proceedings of the 22nd International Workshop on Description Logics (DL 2009), Oxford, UK, July 27-30, 2009, volume 477 of *CEUR Workshop Proceedings*, CEUR-WS.org, 2009.

[30] J. Noessner, M. Niepert, ELOG: A probabilistic reasoner for OWL EL, in: Web Reasoning and Rule Systems - 5th International Conference, RR 2011, Galway, Ireland, August 29-30, 2011. Proceedings, volume 6902 of *Lecture Notes in Computer Science*, Springer, 2011, pp. 281–286. doi:10.1007/978-3-642-23580-1_25.

[31] H. M. Qureshi, W. Faber, Evaluating datalog tools for meta-reasoning over OWL 2 QL, Theory Pract. Log. Program. 24 (2024) 368–393. doi:10.1017/S1471068424000073.

[32] J. P. Balhoff, B. M. Good, S. Carbon, C. Mungall, Arachne: an OWL RL reasoner applied to gene ontology causal activity models (and beyond), in: Proceedings of the ISWC 2018 Posters & Demonstrations, Industry and Blue Sky Ideas Tracks co-located with 17th International Semantic Web Conference (ISWC 2018), Monterey, USA, October 8th - to - 12th, 2018, volume 2180 of *CEUR Workshop Proceedings*, CEUR-WS.org, 2018.

[33] R. Mo, D. Wang, W. Zhan, Y. Jiang, Y. Wang, Y. Zhao, Z. Li, Y. Ma, Assessing and analyzing the correctness of GitHub Copilot's code suggestions, ACM Trans. Softw. Eng. Methodol. 34 (2025) 194:1–194:32. doi:10.1145/3715108.

[34] A. Ziegler, E. Kalliamvakou, X. A. Li, A. Rice, D. Rifkin, S. Simister, G. Sittampalam, E. Aftandilian, Measuring GitHub Copilot's impact on productivity, Commun. ACM 67 (2024) 54–63. doi:10.1145/3633453.

[35] A. Sobo, A. Mubarak, A. Baimagambetov, N. Polatidis, Evaluating LLMs for code generation in HRI: A comparative study of ChatGPT, Gemini, and Claude, Appl. Artif. Intell. 39 (2025). doi:10.1080/08839514.2024.2439610.

[36] A. Sahbi, C. Alec, P. Beust, Semantic vs. LLM-based approach: A case study of KOnPoTe vs. Claude for ontology population from French advertisements, Data Knowl. Eng. 156 (2025) 102392. doi:10.1016/J.DATAK.2024.102392.

[37] V. S. A. Duvvuru, B. Zhang, M. Vierhauser, A. Agrawal, LLM-agents driven automated simulation testing and analysis of small uncrewed aerial systems, in: 47th IEEE/ACM International Conference on Software Engineering, ICSE 2025, Ottawa, ON, Canada, April 26 - May 6, 2025, IEEE, 2025, pp. 385–397. doi:10.1109/ICSE55347.2025.00223.

[38] M. Kim, T. Stennett, S. Sinha, A. Orso, A multi-agent approach for REST API testing with semantic graphs and LLM-driven inputs, in: 47th IEEE/ACM International Conference on Software Engineering, ICSE 2025, Ottawa, ON, Canada, April 26 - May 6, 2025, IEEE, 2025, pp. 1409–1421. doi:10.1109/ICSE55347.2025.00179.

[39] W. Ma, D. Wu, Y. Sun, T. Wang, S. Liu, J. Zhang, Y. Xue, Y. Liu, Combining fine-tuning and LLM-based agents for intuitive smart contract auditing with justifications, in: 47th IEEE/ACM International Conference on Software Engineering, ICSE 2025, Ottawa, ON, Canada, April 26 - May 6, 2025, IEEE, 2025, pp. 1742–1754. doi:10.1109/ICSE55347.2025.00027.

[40] K. Ke, Niodebugger: A novel approach to repair non-idempotent-outcome tests with LLM-based agent, in: 47th IEEE/ACM International Conference on Software Engineering, ICSE 2025, Ottawa, ON, Canada, April 26 - May 6, 2025, IEEE, 2025, pp. 1014–1025. doi:10.1109/ICSE55347.2025.00226.

[41] I. Bouzenia, P. T. Devanbu, M. Pradel, Repairagent: An autonomous, LLM-based agent for program repair, in: 47th IEEE/ACM International Conference on Software Engineering, ICSE 2025, Ottawa, ON, Canada, April 26 - May 6, 2025, IEEE, 2025, pp. 2188–2200. doi:10.1109/ICSE55347.2025.00157.

[42] F. Batole, D. O'Brien, T. N. Nguyen, R. Dyer, H. Rajan, An LLM-based agent-oriented approach for automated code design issue localization, in: 47th IEEE/ACM International Conference on Software Engineering, ICSE 2025, Ottawa, ON, Canada, April 26 - May 6, 2025, IEEE, 2025, pp. 1320–1332. doi:10.1109/ICSE55347.2025.00100.

[43] F. Lin, D. J. Kim, T. Chen, SOEN-101: code generation by emulating software process models using large language model agents, in: 47th IEEE/ACM International Conference on Software Engineering, ICSE 2025, Ottawa, ON, Canada, April 26 - May 6, 2025, IEEE, 2025, pp. 1527–1539. doi:10.1109/ICSE55347.2025.00140.

[44] B. Katzman, D. Tang, A. Santella, Z. Bao, AceTree: a major update and case study in the long

term maintenance of open-source scientific software, BMC Bioinform. 19 (2018) 121:1–121:7. doi:10.1186/S12859-018-2127-0.

[45] B. Motik, R. D. C. Shearer, I. Horrocks, Optimized reasoning in description logics using hypertableaux, in: Automated Deduction - CADE-21, 21st International Conference on Automated Deduction, Bremen, Germany, July 17-20, 2007, Proceedings, volume 4603 of *Lecture Notes in Computer Science*, Springer, 2007, pp. 67–83. doi:10.1007/978-3-540-73595-3_6.

[46] B. Glimm, I. Horrocks, B. Motik, G. Stoilos, Z. Wang, HermiT: An OWL 2 reasoner, J. Autom. Reason. 53 (2014) 245–269. doi:10.1007/S10817-014-9305-1.

[47] N. Matentzoglu, B. Parsia, Ore 2014 reasoner competition dataset, 2014.

[48] N. Matentzoglu, B. Parsia, Ore 2015 reasoner competition corpus, 2015.

[49] R. Sieve, Additional Material for the OxidOWL Reasoner, 2026. URL: https://doi.org/10.5281/zenodo.19627688. doi:10.5281/zenodo.19627688.

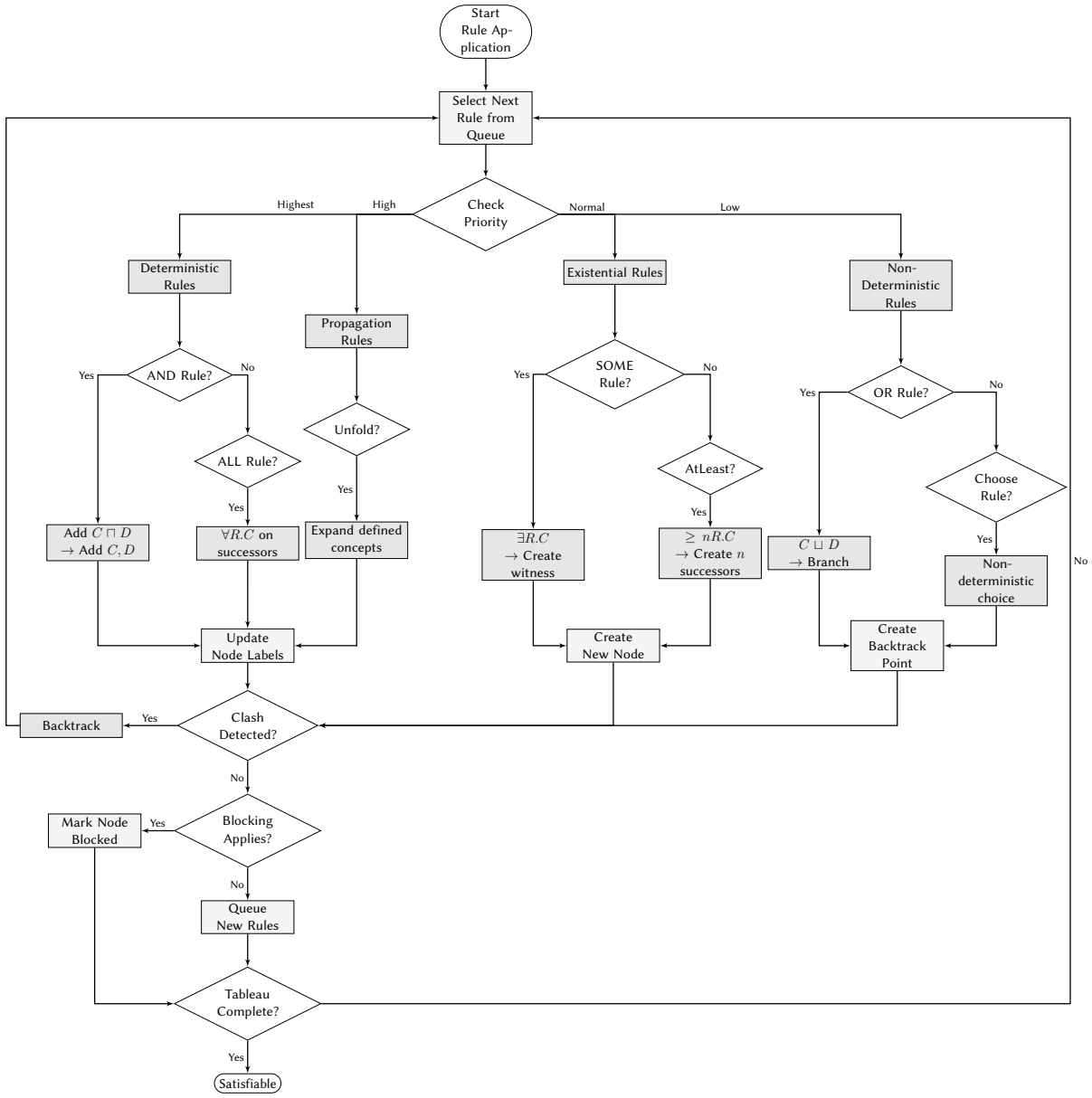


Figure 6: Completion Rule flowchart for the OxidOWL reasoner

A. More Details on OxidOWL

During the development of OxidOWL, we explored various design choices and implementation strategies to improve performance and usability. This appendix discusses the technical aspects of OxidOWL, including its architecture, algorithms, and data structures.

A.1. Completion Rule

Figure 6 illustrates the completion rule applied in OxidOWL during the execution of the Tableau algorithm. The figure provides a visual representation of how the completion rule operates within the system, highlighting the key components and their interactions. Specifically, the algorithm checks for applicable rules and applies them to extend the tableau, ensuring that all necessary conditions are met for successful reasoning. It further checks whether some blocking strategies need to be applied, and marks such nodes as blocked. It further checks for clashes in the tableau, which would indicate contradictions in the logical statements being processed.

A.2. Implementation Details

As mentioned in Sect. 3.1, OxidOWL is implemented in Rust, leveraging its strong type system and performance capabilities. The core components of the source code of OxidOWL are structured across the following folders:

- **Core Reasoning Engine** implements the reasoning algorithms.
- **Ontology Management** handles the manipulation of ontologies.
- **Input Parsers** supports various ontology formats, including OWL and RDF.
- **Query Engine** handles queries over the ontology.
- **SWRL Support** enables reasoning with SWRL rules.
- **OWL 2 Profile Support** provides reasoning for OWL 2 profiles such as $\mathcal{EL}$, $\mathcal{QL}$, and $\mathcal{RL}$.
- **Server** allows for access to the reasoning engine for query execution.
- **Distributed Reasoning** supports distributed computation for large ontologies.
- **DL Clause Generation** converts ontological axioms into DL clauses for reasoning.
- **Import Resolution** manages ontology imports and dependencies.
- **OWL/RDF Semantics** implements the formal semantics of OWL and RDF.
- **OWL 2 DL Validation** validates ontologies against OWL 2 DL constraints.

The modular design of OxidOWL allows for easy extension and integration. The code can be found on GitHub at https://github.com/sievericcardo/oxidowl. The various stages of development and maintenance are highlighted through the various pull requests in which the various features and bug fixing were tackled over time.

A.3. Reasoner Flow

The reasoning flow in OxidOWL is illustrated in Fig. 7. The process begins with the loading of an ontology through the Public API Layer, which then passes the ontology to the Reasoner Core. The core component orchestrates the reasoning tasks by invoking the appropriate tableau algorithms, data management, and strategies.

The tableau algorithms process the ontology, applying the necessary reasoning strategies to derive logical consequences. The core components manage the various aspects of the reasoning process, ensuring that blocking, completion, expansion, and dependency tracking are handled effectively. It further employs caching systems to optimise performance during reasoning tasks. We used Least Recently Used (LRU) eviction policy to manage memory usage effectively.

To further enhance the reasoning process, we employed a set of deterministic rules (e.g. conjunction, subclass, equivalence„ etc.) within the saturation engine. These conditions are pre-computed to optimise the application of expansion rules during reasoning, reducing unnecessary computations and improving overall efficiency. The saturation engine applies the rules to the ontology before invoking the tableau algorithms. By doing so, it reduces the number of inferences that need to be handled by the tableau, which is typically more computationally intensive. By appropriately marking concepts based on their saturation status, the reasoner can make informed decisions about when to escalate to full tableau reasoning. It can further support incremental reasoning by re-saturating only affected concepts when the ontology changes, avoiding full recomputation.

B. Input used for the development of OxidOWL

To ensure that the agent could have complete overview of the standards, features, and available sources that were relevant for the development process, the following links were provided to the agent

- OWL - https://www.w3.org/TR/owl-features/
- OWL 2 - https://www.w3.org/TR/owl2-primer/

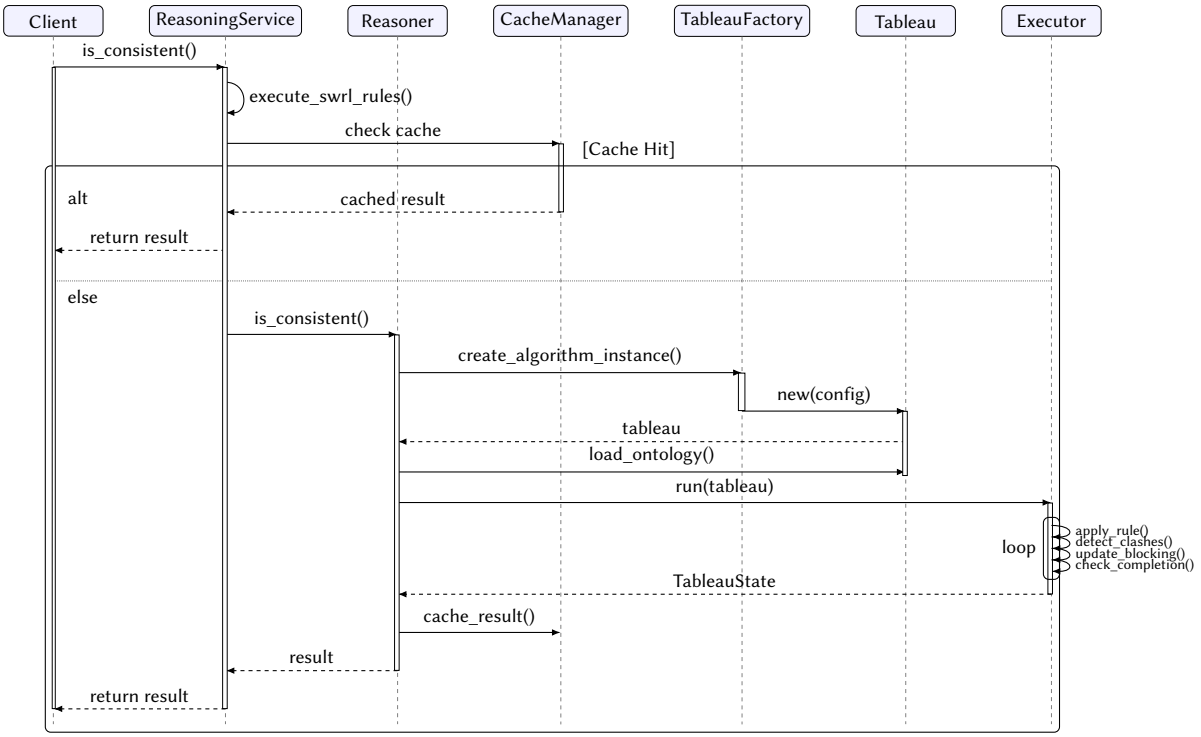


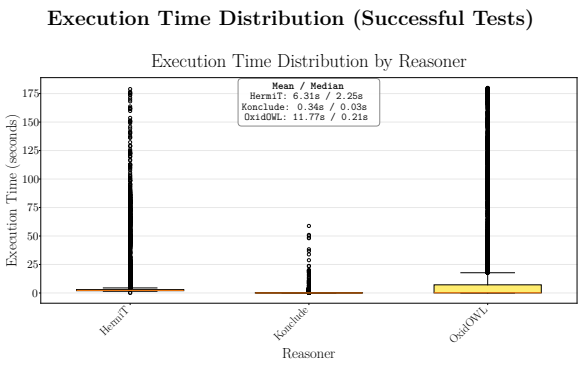
Figure 7: Reasoning flow for OxidOWL. The diagram illustrates the interactions between the main components during a reasoning task, highlighting the cache hit and miss scenarios employed for improved performance. The loop within the Executor represents the iterative application of tableau rules until completion. They are employed to apply the different tableau rules, detection of clases in the reasoning process, update the blocking strategy, and check for completion.

- OWL 2 overview - https://www.w3.org/TR/owl2-overview/
- OWL 2 mapping to RDF - https://www.w3.org/TR/2012/REC-owl2-mapping-to-rdf-20121211/
- OWL 2 XML serialisation - https://www.w3.org/TR/2012/REC-owl2-xml-serialization-20121211/
- OWL 2 functional - https://www.w3.org/TR/2012/REC-owl2-syntax-20121211/
- OWL 2 Manchester - https://www.w3.org/TR/2012/NOTE-owl2-manchester-syntax-20121211/
- OWL 2 RDF - https://www.w3.org/TR/2012/REC-owl2-mapping-to-rdf-20121211/
- OWL 2 Turtle - https://www.w3.org/TeamSubmission/turtle/
- RDF semantics - https://www.w3.org/TR/rdf11-concepts/
- RDF - https://www.w3.org/RDF/
- RDFS - https://www.w3.org/TR/rdf-schema/
- SWRL - https://www.w3.org/submissions/SWRL/
- HermiT - https://github.com/owlcs/hermit-reasoner
- Konclude - https://github.com/konclude/Konclude

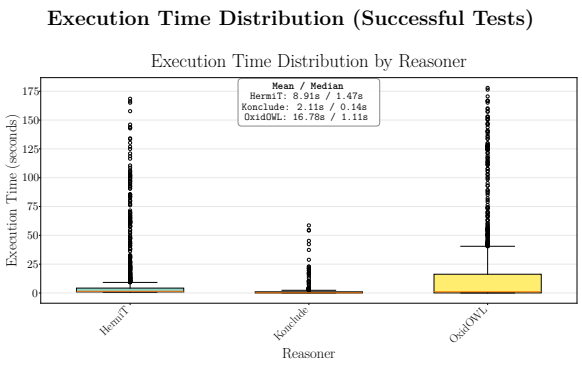
Adding them to the context allows coding agents to have a more thorough understanding of the information required to complete the task as requested. We uploaded on Zenodo [49] the different plans generated for the implementation of new features and maintenance of OxidOWL, as well as some of the documents that were asked to the agent to produce to document the development, testing, and improvement process. However, since we started with a different methodologically approach, we lost some of the data regarding the initial development prompts.

C. Additional Material Provided as Input to Coding Agents

As mentioned in Sect. 3.3, we can construct the prompt to feed to GitHub Copilot by means of planning. Through successive interactions, the plan component generates a Markdown file containing the whole



(a) Execution time distribution comparison of HermiT, Konclude, and the OxidOWL reasoner on the subset of the ORE 2014 competition dataset.



(b) Execution time distribution comparison of HermiT, Konclude, and the OxidOWL reasoner on the ORE 2015 competition dataset.

Figure 8: Comparison benchmark between HermiT, Konclude, and OxidOWL. (a) shows the result with the ORE 2014 dataset (b) shows the result with the ORE 2015 dataset.

prompt. However, we further improve the code generation process by feeding instructions to GitHub Copilot as a Markdown file. In the file, we specify programming aspects that we want to be enforced when proposing a technical solution. Among these, we specify:

- *Overview* where we provide a high-level overview of the project and the key concepts.
- *Core Responsibilities* for the coding agents, detailing the aspect that must be followed (e.g. code generation, code review, optimisation, testing, etc.).
- *Communication Style* to enforce certain patterns during the agents execution; we can request to provide clear explanation for the different choices taken.
- *Coding Standards* where we can enforce certain aspects related to the programming language (e.g. avoiding the usage of unchecked `unwrap`).
- *General Rules* that have to be observed at all time, either technical (avoid `unsafe` in Rust), or general (e.g. "Never use deprecated library or unstable features without proper justification")

It is also useful to provide an example for the task to be executed for the agents to follow. For example, we can provide the following

1. Understand the Problem: Restate what's being asked.
2. Propose a Solution: Outline approach and reasoning.
3. Implement the Code: Provide idiomatic Rust implementation.
4. Explain the Design: Describe trade-offs, safety guarantees, and performance characteristics.
5. Next Steps: Suggest how the user might test or extend the solution.

By feeding the instruction file, we can improve the overall development by providing certain conditions and standards that we want to be enforced during the development process.

D. Comparison of OxidOWL with Other Reasoners

To further compare and contrast the performance of OxidOWL with HermiT and Konclude, we tested all three reasoners against a subset of classification and consistency tasks of ORE 2014 dataset and over all the ORE 2015 dataset. The tests were conducted using the HermiT binary[10] and the installed version of Konclude[11]. We used binaries for all three reasoners to ensure similar conditions and to compare the three reasoners.

[10] http://www.hermit-reasoner.com/download.html
[11] https://github.com/konclude/Konclude

Table 3
Breakdown of the lines of code and number of modules for each main component of OxidOWL.

Component	Lines of Code	Number of Files
Core Engine	23227	43
Reasoner	1307	5
Saturation Engine	4321	7
Tableau Algorithm	5801	12
Hypergraph	1486	2
Query	18456	21
Parsers	11086	9
Ontology	5186	7
SWRL Support	13558	26

Following the tests conducted in Sect. 5, we applied a timeout of 180 seconds for these tests as well. We then used the results to compare the execution time for the different reasoners within the different datasets. As shown in Fig. 8, the overall performance of OxidOWL is comparable in terms of execution. While Konclude was still the fastest of the three, the difference, in terms of median value, OxidOWL showed to be a viable alternative.

E. Development and Maintenance Timeline of OxidOWL

In Tbl. 3, we provide a breakdown of the lines of code and number of modules for each main component of OxidOWL, giving an overview of the size and complexity of the different components. The core engine is the largest component in terms of lines of code, followed by the query engine and SWRL support. The number of modules also varies between components, with the core engine having the most modules, reflecting its central role in the reasoning process. This breakdown provides insight into the structure and organisation of the OxidOWL codebase.

The development of OxidOWL started in mid 2025, with the initial implementation of the core reasoning engine and basic support for OWL 2 DL. The first five versions were not initially fully released via GitHub release, but rather developed iteratively with regular commits and pull requests to the GitHub repository. The initial development phase focused on implementing the core reasoning algorithms and ensuring compliance with OWL 2 DL standards, followed by several updates that added support for additional features such as SWRL rules and OWL 2 profiles.

The first official release of OxidOWL was made available in November 2025, marking a significant first step in the project, where most of the core features were implemented and tested, and `unwrap` calls present were removed from the codebase.

The maintenance phase has been ongoing since the initial release, with regular updates to fix bugs, improve performance, and add new features and evolving standards. The development timeline is documented through the various pull requests and commits on the GitHub repository, providing a clear history of the project's evolution.

The overall development and maintenance process has spanned for a year, with continuous improvements and updates to ensure that OxidOWL remains a robust and efficient reasoner for OWL 2 DL ontologies. However, since the codebase has been generated by coding agents, the development process has been more iterative and less linear than traditional software development, with multiple features being developed in parallel and integrated over time.

The actual time spent developing and maintaining OxidOWL is difficult to estimate, as the coding agents worked in the background with the review happening after they finished their task. However, the overall timeline of the project, from initial development to the current state, is still estimated to be less than three months of work. This time also considers the time taken for the initial design time and planning phase.