A Neuro-Symbolic Approach for Test Oracle Generation

Doctoral Dissertation submitted to the
Faculty of Informatics of the *Università della Svizzera italiana*
in partial fulfillment of the requirements for the degree of
Doctor of Philosophy

presented by
Davide Molinelli

under the supervision of
Prof. Mauro Pezzè
co-supervised by
Dr. Luca Di Grazia, Dr. Alberto Martín López

December 2025

Dissertation Committee

Prof. Gabriele Bavota	Università della Svizzera Italiana, Switzerland
Prof. Carlo Alberto Furia	Università della Svizzera Italiana, Switzerland
Prof. Annibale Panichella	Delft University of Technology
Prof. Mike Papadakis	University of Luxembourg

Dissertation accepted on 04 December 2025

Prof. Mauro Pezzè
Research Advisor
Università della Svizzera italiana, Switzerland

Dr. Luca Di Grazia
Research Co-Advisor
University of St. Gallen, Switzerland

Dr. Alberto Martín López
Research Co-Advisor
Università della Svizzera italiana, Switzerland

Prof. Walter Binder/ Prof. Silvia Santini
Prof. Walter Binder / Prof. Stefan Wolf

I certify that except where due acknowledgement has been given, the work presented in this thesis is that of the author alone; the work has not been submitted previously, in whole or in part, to qualify for any other academic award; and the content of the thesis is the result of work which has been carried out since the official commencement date of the approved research program.

Davide Molinelli
Lugano, 04 December 2025

A mia madre e mio padre,
È il mio ringraziamento per tutti i sacrifici fatti per me.

A Luigi,
che da ragazzo saltava la scuola e da nonno
mi ha insegnato quanto importante fosse l'istruzione.

A Sofia,
Per l'amore ricevuto, che non si dimentica.
Ho mantenuto la promessa.

È una di quelle cose che è meglio che non ci pensi, se no ci esci matto. Quando cade un quadro. Quando ti svegli un mattino, e non la ami più. Quando apri il giornale e leggi che è scoppiata la guerra. Quando vedi un treno e pensi io devo andarmene da qui. Quando ti guardi allo specchio e ti accorgi che sei vecchio. Quando, in mezzo all'Oceano, Novecento alzò lo sguardo dal piatto e mi disse: "*A New York, fra tre giorni, io scenderò da questa nave*". Ci rimasi secco. Fran.

Novecento. Un monologo,
Alessandro Baricco

Abstract

Automatic generation of test oracles is an open challenge in software testing. Although the generation of test prefixes has received substantial attention, the oracle problem, defined as the automatic generation of assertions that verify program behavior, remains largely open. This dissertation proposes the integration of symbolic and neural techniques in a unified neuro-symbolic framework to effectively generate test oracles.

The research conducts a systematic three-stage empirical study spanning the generation of both axiomatic and concrete test oracle. The first stage defines TRATTO, a neuro-symbolic approach that generates axiomatic test oracles token-by-token. TRATTO integrates a symbolic component that constrains the search space of valid tokens to ensure compilability, and a neural component that guides the token generation toward semantically relevant oracles. TRATTO achieves 73% accuracy, 72% precision, and 61% F1-score on a ground-truth dataset, outperforming the symbolic baseline Jdoctor (61% accuracy, 62% precision, 25% F1-score) and the neural baseline GPT-4 (40% accuracy, 24% precision, 37% F1-score). The approach generates three times more correct oracles than Jdoctor while producing ten times fewer false positives than GPT-4, demonstrating improved soundness through neuro-symbolic integration.

The second stage is a large-scale empirical study to evaluate large language models for generating concrete oracles. We conducted the study with an unbiased dataset of 13,866 test oracles from 135 Java projects, with all test cases created after the models' training cut-off date to ensure strict separation from training data. The study systematically varies model families, sizes, and prompt configurations, and generates 554,640 predictions. The results show that larger models consistently outperform smaller ones across all specializations, and that providing additional contextual information beyond the test prefix and the source code of the invoked methods do not lead to any significant performance improvement. Mutation analysis shows that LLM-generated oracles achieve fault-detection capability comparable to developer-written oracles (43% vs. 45% mutation score), while also revealing a critical limitation: 50% of the generated oracles in the experiment fail to compile or execute.

The third stage defines TRACTO, a neuro-symbolic approach for concrete oracle generation, composed of a symbolic module that integrates a grammar-based static analysis and a project identifiers resolution system to retrieve candidate tokens and enforce validation constraints, and a neural module that performs token selection and concrete literal inference. The systematic comparison against a fine-tuned pure neural model (Qwen2.5-Coder-3B) and the best performing vanilla model of the empirical study (Qwen2.5-Coder-32B) on 3,448 post-cutoff test oracles reveals fundamental trade-offs. The pure neural model achieves the highest raw accuracy (33%) compared to TRACTO (20%) and the vanilla LLM (10%). However, TRACTO yields more robust oracles, reaching higher compilation rates (80%) over the pure neural model (73%) and the vanilla LLM (10%) and higher test-pass rates (59% over 51% and 7%, respectively).

The results presented in this dissertation support the research hypothesis that a neuro-symbolic approach improves the quality of the generated oracles with respect to purely neural approaches. Symbolic constraints reduce false positives, improve compilation rates, and enhance test pass rates across both axiomatic and concrete oracle generation. However, these benefits manifest alongside reduced test oracles generation coverage due to early termination when validation constraints detect problematic patterns.

Acknowledgements

Ho sempre scandito la mia vita a suon di date. Non è un'ossessione. Non è un superpotere. Funziono così. Ricordo a distanza di anni quando ho fatto il test di ingresso per l'ammissione al Politecnico di Milano. Era sabato 13 luglio 2013 (è solo un esempio). Ricordo l'esame di Fisica 1. Lo scritto era sabato 19 luglio 2014. L'orale, venerdì 25 luglio (è solo un altro esempio). Ricordo (e qui mi fermo con gli esempi) il giorno in cui ho avuto la prima call con il Professor Mauro Pezzè, per conoscerci e approfondire il desiderio di iniziare un dottorato. Era il 30 Novembre 2019. Esisteva ancora Skype. Se ripenso a quella sera, nell'ufficio di papà, ormai a tarda sera, mi viene in mente un'emozione più forte delle altre. Era la paura. Mauro ha avuto pazienza con me. E prima di ringraziarlo per il percorso fatto insieme, devo ringraziarlo per la premura che ha avuto nei miei confronti: mi ha lasciato il tempo di decidere quale fosse la scelta migliore per me e mi ha aperto alla possibilità di vivere un percorso che mi ha fatto crescere come ricercatore e come uomo. Mauro, ti ringrazio personalmente per avermi accompagnato in questi anni. Non mi sono mai sentito solo, anche quando nei momenti difficili c'era prima di tutto un amico, prima che un professore, a confortarmi per ogni *insuccesso* che ha caratterizzato il mio percorso. Ti ringrazio per avermi trasmesso la passione per la ricerca e aver messo a disposizione tutta la tua esperienza e professionalità perché potessi portare, infine, a termine con *successo* questo percorso. Non posso che essere riconoscente di tutto ciò che ho ricevuto. Mi sono sentito libero di fare ricerca e cambiare quando ho ritenuto che fosse necessario. Non c'é nulla di scontato in tutto questo. Ho avuto il piacere e l'onore di poter collaborare ed essere guidato da una figura rispettata dal mondo accademico per i successi ottenuti nel corso della sua carriera.

Ringrazio il Dottor Alberto Martín López per avermi fatto appassionare alle sue idee di ricerca al punto da condurmi a farle diventare anche un po' mie. Lo ringrazio per essere stato un esempio di professionalità e ambizione. Per essere stato presente quando avevo bisogno di supporto sia a livello professionale che personale, e rude quando era necessario mi guardassi dentro e riconoscessi gli errori commessi (sempre in maniera corretta e per il bene mio e del gruppo).

Ringrazio il Dottor Luca Di Grazia per essersi inserito nel mio lavoro di dottorato e aver fornito un contributo significativo ai risultati della nostra ricerca. Lo ringrazio per i modi gentili e la discrezione che ha sempre avuto non solo nei miei confronti, ma nei confronti di tutto il gruppo. Nei mesi in cui ha collaborato con noi ha messo a disposizione le sue capacità per essere di supporto a tutti, dal professor Mauro all'ultimo studente più giovane.

Alberto e Luca, ammiro profondamente la vostra dedizione al lavoro e vi auguro una carriera di successo perché meritate che il vostro lavoro e i vostri sacrifici vengano riconosciuti e ricompensati.

Ringrazio il gruppo di ricerca, composto da Noura, Alind, Niccolò, Ketai e Fabio per aver

condiviso con me questo percorso da studenti di dottorato. La distanza non ci ha aiutato a vivere la quotidianità insieme, ma ho sentito la vostra presenza in ogni messaggio di sostegno a fronte di ogni difficoltà, e di congratulazioni a seguito di ogni risultato raggiunto.

Sebbene viviamo la maggior parte della nostra quotidianità nel nostro ambiente di lavoro, La vita però è anche e soprattutto ciò che ci circonda quando torniamo a casa stanchi dall'ufficio. Dicono che sia importante circondarsi delle persone giuste nella vita. Non c'é cosa più vera.

Ringrazio Alessio, Emanuele e Silvio per non esserci mai persi davvero, anche dopo esserci allontanati geograficamente. Le occasioni per vedersi sono sempre rare, ma ogni volta che vi vedo riabbraccio degli amici con cui ho stretto un rapporto speciale. Alessio, mi hai accompagnato in questo percorso in ogni messaggio che non hai mancato di inviarmi per sapere come stessi e come procedesse il dottorato. Sei stato una spalla sia nel contesto lavorativo che in quello personale. Mi sei stato vicino in ogni momento di questo mio percorso. Emanuele, ricordo ancora quando dovevo decidere se intraprendere questa avventura in un posto lontano da casa. Mi hai detto: ricordati che il cambiamento è ciò che spaventa di più le persone, ma è anche la cosa che le spinge verso qualcosa di migliore. Hai contribuito a spingermi a fare la scelta più giusta per la mia crescita personale e professionale. Silvio, ti ho sempre ammirato per la tua intelligenza e per la tua capacità di analizzare le cose in maniera molto obbiettiva. Hai sempre avuto grandi ambizioni e sono sempre rimasto affascinato dalla tua ricerca di migliorare e diventare la versione migliore di te stesso, sia nel contesto lavorativo che privato. In questi anni ho conosciuto un tuo lato umano che non ho avuto il piacere di osservare negli anni dell'università. È stato davvero bello potersi aprire, confrontare e rafforzare la nostra amicizia. Rivolgendomi a tutti e tre, posso dire che rimanete il successo più grande ottenuto durante il mio percorso di formazione al Politecnico di Milano. Non ricordo i voti. Ricordo i progetti e il tempo speso insieme a preparare gli esami.

Ringrazio Giacomo: l'amicizia costruita ai tempi della triennale non è mai cambiata. Nonostante non ci vediamo da tempo e ci sentiamo sporadicamente rimani una persona su cui posso sempre contare per una telefonata, un consiglio, una risata. Non abbiamo mai perso lo spirito di prenderci in giro reciprocamente e sono contento tu sia parte della mia vita. Dovevi essere in questa pagina.

Ringrazio Nicholas, Alessandra, Simone, Francesca, Mirko e Chiara per le serate trascorse insieme, le risate, i momenti filosofici e di riflessione. Sono riconoscente di ogni momento che abbiamo condiviso. Nicholas, sei stato una delle persone che mi è stata più vicina in questi anni. Mi hai spinto a prendere la decisione di fare un'esperienza all'estero. Mi sei stato vicino nei momenti difficili di vita personale e ti sei aperto raccontandomi di te, da vero amico. Hai realizzato il mio sogno di vedere Parigi. Sono felice di averla vista con te. Alessandra, io e te non ci siamo mai detti tante cose carine. Siamo due vergini che per orgoglio non farebbero mai il primo passo per dirsi qualcosa di sdolcinato. E va bene così. Mi sento di farne uno io qui: ti voglio bene. A volte non ci capiamo (spesso), a volte ci capiamo un po' di più (poco). Ma so di poter contare su di te come amica. Sono fiero delle tue battaglie, perché ti esponi con le tue idee. Sei la ragazza che scende in piazza, se serve. E indipendentemente dalle proprie convinzioni, mi hai insegnato con il tempo che se non ci si espone, le cose non possono mai davvero cambiare. Francesca, sei l'amica a cui posso confidare ogni paura ed emozione che mi frulla nel cuore, mentre sorseggiamo una tazza di té. In questi anni ne abbiamo assaporati di tutti i gusti più uno. A te non mi sembra di doverti dire che ti voglio bene. Te l'ho detto tante volte. Già lo sai. Chiara, ci sentiamo sempre troppo poco, ma nei momenti in cui serve non manchi di esserci e supportarmi. Mi hai raccontato tanto di te stessa in questi anni e mi hai aiutato tanto a vedere le cose dalla giusta prospettiva, senza essere giudicante.

Ringrazio Andrea, Gabriele, Danae, Simone, Lorenzo, Gaia, Giorgia. Siete il gruppo di cui ho bisogno ogni volta che sento di voler ridere fino ad avere il mal di pancia. Mi siete stati vicini sin dal primo giorno di questo percorso, quando avevo bisogno del vostro sostegno per sentirmi un po' meno lontano da casa. Andrea e Lorenzo, i ricordi di Lottstetten sono indimenticabili. Non so se ho scelto gli amici più affidabili per trovare casa. Sicuramente ho trovato quelli giusti per portarmi la schiscietta a lavoro e trasformare una settimana ordinaria in una leggendaria. Gabriele e Danae, il papà e la mamma del gruppo. Mi siete stati sempre vicino e siete sempre stati presenti a vostro modo, come quando avete voluto accompagnarmi durante la seconda settimana di lavoro per non farmi stare solo e aiutare ad ambientarmi. Non lo dimentico. Avete compreso i momenti in cui sono stato assente per tanto tempo, con i miei silenzi, vuoi per dei pensieri, vuoi per i miei impegni, vuoi perché mi perdo tante volte in un bicchier d'acqua. Mi avete sempre capito e mi avete sempre accolto con un sorriso e un abbraccio, nonostante tutto. Per voi l'importante è sempre stato che stessi bene. Grazie di cuore.

Ringrazio Simone, un amico che ho avuto modo di scoprire appieno in questi anni dentro e fuori dal campo di atletica. Sei stato di supporto quando stavo crollando con la testa. Mi hai aiutato ad affrontare le difficoltà con lo spirito sportivo che ci ha sempre caratterizzato. Sei stato una valvola di sfogo sia per la mente che per il fisico. Mi hai aiutato a non mollare e rialzare la testa. Ti ho sempre ammirato per la tua caparbietà e le tue capacità. Hai raggiunto dei traguardi incredibili e sei stato di esempio e d'ispirazione per tutti questi anni. Modello di disciplina e serietà. Sono fiero di te e felice di aver consolidato il nostro rapporto.

Ringrazio Giulia, un'amica speciale che ho conosciuto all'inizio di questo percorso e che è diventata un riferimento nella mia vita. Sei stata la persona che più mi ha fatto compagnia i primi anni del mio dottorato. La persona più buona e innocente che abbia mai avuto la fortuna di poter incontrare. Con te mi sono potuto aprire completamente e confidare per ogni situazione. Nonostante vivessimo in due paesi diversi, nonostante avessimo vissuto insieme con gli amici soltanto una settimana in Toscana, siamo riusciti a creare un legame forte e sincero. Sono fiero di noi.

Ringrazio mia cugina Elisabetta. Sei unica. Mi hai accompagnato in ogni momento di questo percorso standomi vicino e aiutandomi ad affrontare i miei problemi più intimi e personali. Sei stata la mia valvola di sfogo quando non sapevo a chi rivolgermi. Ti sei presa cura di me quando mi sentivo giù di morale. Non hai mai mancato di assicurarti che stessi bene e che fossi sereno. Sono contento di aver potuto vivere con te una notte speciale a San Siro. Non la dimenticherò mai. Ti voglio bene.

Ringrazio mia cugina Chiara, che mi ha accompagnato dall'altra parte del mondo alla mia ultima conferenza, prima della chiusura di questo dottorato. Mentre scrivo queste parole, stai dormendo accanto a me e forse mi è più facile dirtelo così: ti ringrazio di cuore per aver reso questo viaggio in Korea speciale. Hai reso tutto più facile con la tua compagnia e il tuo umorismo. Non avrei potuto chiedere di chiudere questo percorso nel modo migliore. Grazie per aver accettato di fare questa pazzia insieme. Ci abbiamo messo 31 anni per fare un viaggio, ma spero tu possa tornare a casa felice di raccontare cosa abbiamo vissuto. Con te, in questi giorni, sono ritornato un po' il Davide bambino che a Tagliolo era felice di poter giocare tutto il giorno con la sua cuginetta che non vedeva mai. Ci siamo fatti un regalo meraviglioso che spero ricorderemo per sempre. Ti voglio bene.

Ringrazio la mia psicologa Camilla, che in questi ultimi anni ha accettato di prendermi per mano e accompagnarmi in un viaggio alla scoperta di me stesso. Penso che la terapia sia stata una delle scelte più coraggiose, difficili, ma anche più importanti che abbia mai intrapreso. Ti ringrazio per avermi accolto e avermi aiutato a diventare la versione migliore di me stesso,

affrontando il mio passato, realizzando quali fossero le mie priorità e aiutandomi a superare le perdite e i lutti dell'ultimo anno. Mi hai ascoltato senza giudicare e mi hai aiutato a affrontare le mie paure. Non ho mai preso un aereo prima di conoscerti. In questo momento mi trovo dall'altra parte del mondo. Direi che di strada ne abbiamo fatta.

Ringrazio nonno Luigi che non si è mai dimenticato di chiedermi come stessi in Svizzera, se fossi contento del mio lavoro e se mi fossi fatto degli amici. Anche nei suoi ultimi giorni di vita. Non sai quanto vorrei poter festeggiare questo traguardo anche con te. Ma so che in qualche modo non mancherai di esserci anche questa volta.

Ringrazio Sofia, una persona che è stata per me fondamentale nel mio percorso, non solo lavorativo, ma anche e soprattutto di vita. Sofia, sei stata una bellissima cometa di 8 mesi, sopra il mio cielo. Ricorderò per sempre la notte di mezza estate quando ti proposi di andare a vedere le stelle cadenti. Quella sera nel cielo brillavano tante stelle sopra la mia testa, ma io avevo occhi solo per una sulla Terra. Sei stato il combustibile di cui avevo bisogno per ottenere i risultati che mi sono a lungo mancati. Mi hai donato l'amore e la serenità necessari per affrontare i momenti difficili con una forza che non trovavo da tempo. Mi piace pensarti come il tovagliolo che il cameriere piega e mette sotto la gamba del tavolo. E il tavolo smette di tremare. Il tavolo ritrova il suo equilibrio. Su questo dottorato c'é anche il tuo piccolo grande contributo. Dicono che il primo amore non si scorda mai. Io il primo l'ho dimenticato grazie alla terapia. Il secondo lo ricorderò con affetto e tenerezza per tutta la vita.

Ringrazio Ilaria, che è entrata nella mia vita una sera di Settembre, durante un viaggio di un treno che non dovevo prendere. Decidere di salirci, è stato una delle decisioni più semplici e belle che potevo fare quest'anno. Ci stiamo ancora conoscendo e non so dove porterà tutto questo, ma sono contento di consegnare questa tesi, mentre tu mi *rompi le scatole* al telefono, per sapere come sto e se ho voglia di condividere con te le mie emozioni del momento. Te le racconterò presto per lettera (come piace fare a noi), ma voglio tu sappia, qui, che sono stanco, ma felice. Ed è anche grazie alla tua compagnia.

In ultimo, sento di ringraziare le persone più importanti della mia vita. Il 13 luglio 2013, fuori da un'aula del Politecnico alla fine dell'esame di ammissione, Mamma aspettava leggendo un libro su una panchina. Voleva essere lì con me. Il 25 luglio 2014, all'uscita del cancello di via Giurati, fuori da un'aula del Politecnico ad aspettarmi in macchina alla fine dell'orale di Fisica 1 c'era Papà. Voleva essere lì con me. Mamma e papà ci sono sempre stati. Forse troppo, a volte. Ma come mi insegna Camilla, nessuno ha mai imparato a fare il genitore prima di diventarlo. Hanno cercato di farlo nel miglior modo che potevano: con amore. E io non posso che essere riconoscente di questo. Mamma, Papà, grazie per ogni sacrificio che avete fatto per me. Mamma, grazie per ogni volta che mi hai accompagnato su un campo di calcio per farmi divertire. Papà, grazie per essermi venuto a prendere ad ogni fine allenamento e non aver mai mancato a una partita, altrimenti avrei giocato male. Grazie per aver pensato alla mia istruzione. Grazie per esservi messi contro di me quando pensavate che qualcosa o qualcuno potesse non essere la scelta giusta per la mia vita. E grazie per avermi spinto a non mollare in questo percorso tanto difficile. Grazie per avermi supportato. Per ogni volta che mi avete accompagnato alla stazione ad ogni partenza, e per ogni volta che mi avete accolto con un sorriso quando tornavo a casa. Ce l'abbiamo fatta. Dimentico sempre di dirvelo. Ora ho l'opportunità più che mai di farlo: vi voglio bene.

Contents

Figures

Tables

Chapter 1

Introduction

Testing assesses the quality of software systems by executing test cases that sample the execution space of the software application. Test cases are composed of a *test prefix* and an *oracle*. The test prefix drives the unit under test to an interesting state through primitive inputs, simulated objects (*mocks*), and sequences of statements. The oracle checks the validity of the states that the application reaches when executing the test case, by asserting conditions that those states must satisfy.

Automatic test case generation represents an important area of research in software testing, as it substantially reduces the overall time and effort spent on the testing process while increasing software quality. Most research on test automation has focused on the automatic generation of test prefixes. The oracle problem, that is, the problem of automatically generating test oracles [4], has received less attention, and remains an open challenge in software testing to a larger extent.

This PhD thesis addresses the problem of automatically generating semantically relevant test oracles by investigating the comparative effectiveness of neuro-symbolic and purely neural approaches.

Most current approaches to automatically generate test oracles produce either *implicit* oracles that check for the presence of general failures (such as program crashes and unhandled exceptions), or *differential* (or *regression*) oracles that compare the output of different program versions. Mature unit test generation tools, such as Randoop [54] and Evosuite [23], generate both test prefixes and implicit or regression oracles. However, implicit and regression oracles are not aware of the intended behavior of the system under test; consequently, they miss many relevant failures that derive from the deviation of actual program behavior from intended program behavior.

Recent symbolic and neural-based approaches have shown potential by exploiting natural language processing (NLP) to understand the semantics of software in the absence of formal specifications. These approaches derive both *assertion oracles* (which check actual output against expected output) and *exception oracles* (which capture the intended exceptional behavior of the program under test) from unstructured sources of information such as code, comments, and documentation.

Both symbolic and neural approaches offer interesting solutions to automatically generate test oracles, however, they suffer from complementary limitations. Symbolic techniques, such as [61, 8, 9], leverage a fixed set of domain-specific rules, for instance, pattern and lexical

matching, and achieve remarkable results both in terms of precision and recall. However, they fail to generalize well when the natural language artifacts fall outside the defined patterns, narrowing their applicability. Neural approaches leverage deep learning and transfer learning to generate either concrete test cases or concrete oracles that depend on the test input [65, 19]. Although they require large amounts of typically labeled data for training, they demonstrate wide applicability and capability in deriving test oracles from ambiguous natural language descriptions. However, neural approaches produce oracles that suffer from high false positive rates—that is, they generate many assertions that do not correspond to the intended behavior of the program.

This PhD thesis explores the integration of symbolic and neural techniques within a unified neuro-symbolic framework for automatically generating test oracles. The main goal of this thesis is to assess whether combining the structural rigor of symbolic reasoning with the adaptability of neural models produces more effective oracles than purely neural methods. By constraining the oracle synthesis process through statically derived program information, such as programming language grammar, type information, and code context, the approach strategically narrows the set of candidate tokens and steers the neural component toward semantically valid and contextually relevant oracles. The methodology targets the generation of both assertion and concrete oracles, with the goal of producing test oracles that are structurally sound, broadly applicable, and precise in capturing intended behavior.

Test oracles can be either *concrete* or *axiomatic*. Concrete oracles are test assertions that predicate on the results expected for the concrete test input. For instance, an assertion can predicate on the result of a method that computes the `sum` between two addends, whose values are `20` and `22` respectively, by asserting that the expected output is `42`. Axiomatic oracles are program assertions that predicate on input or output parameters of the program. For instance, the output of a method that computes the rest of a division between two input numbers `num1` and `num2` will always be greater than or equal to `0` and less than the value of the first dividend. Axiomatic oracles generalize concrete oracles and are suitable for *property-based* or *parameterized* test cases, that check the desired behavior of a system on properties that always hold, or that hold for a well-defined range of the input space, independently from the concrete value of the input parameters. Although both types of oracles ultimately serve the same purpose, namely detecting deviations of a program unit from its expected behavior, their capabilities can be considered supplementary. Concrete oracles tend to outperform axiomatic oracles in fault detection due to their precision and specificity; axiomatic oracles are inherently more expressive and general. Their ability to model invariants over a wide input space makes them more adaptable across test-related contexts, such as program comprehension, requirements specification or runtime verification.

This PhD work investigates whether combining symbolic and neural methods improves the generation of test oracles. It conducts a comparative analysis of neuro-symbolic and purely neural techniques across both axiomatic and concrete oracle types. Through systematic empirical evaluation, the study shows the benefits and limitations of integrating symbolic constraints with neural generation for automated oracle creation.

1.1 Research Hypothesis and Contribution

Outlined the research context and the stated problem, the dissertation argues that:

Thesis Statement

Integrating the precision of symbolic testing techniques with the generalization capabilities of neural testing methods leads to the generation of semantically relevant test oracles with improved soundness compared to purely neural approaches.

Building on recent advances in neural models for code and documentation understanding, this dissertation investigates whether neuro-symbolic approaches can infer semantically relevant oracles , while restraining imprecision and hallucinations that arise from the purely probabilistic and statistical generative AI and represents a critical issue in software testing [35, 21].

This dissertation conducts a systematic comparison to understand the relative strengths and limitations of neuro-symbolic and purely neural methods through a structured, three-stage empirical investigation that reflect both the methodological requirements and the evidence gathered along the way.

We develop a neuro-symbolic method to generate axiomatic test oracles, leveraging a symbolic module that bounds the search space and reduces the false positives while allowing a neural model to generalize over noisy or incomplete specifications. We target axiomatic oracles to infer general and reusable test assertions, while limiting the generation of specific literals (numbers, strings) that are difficult to validate statically and can inflate false positives when guessed by a model.

We assess both strengths and limitations of a neuro-symbolic approach, by evaluating the performance against prior neural and symbolic approaches. Our experiments indicate that axiomatic oracles suite well runtime verification, while concrete oracles are effective at exposing faults in testing, although they are prone to spurious assertions.

We conduct an experimental study that investigates LLMs for generating concrete test oracles. We systematically vary the model family, type and size, the prompt design, and the scope of contextual information to quantify their impact on the generation of meaningful and precise concrete test oracles on an unbiased, post-cutoff dataset.

The thesis provide a benchmark that highlights both the strengths and limitations of purely neural methods. These insights inspire the design of a neuro-symbolic approach: we relax the restriction on literals to improve fault detection, while retaining symbolic control over syntax, typing, and oracle structure to reduce false positives.

We evaluate the neuro-symbolic approach for concrete test oracles generation in direct comparison with purely neural models fine-tuned to infer the same oracles. This evaluation tests the research hypothesis, and assesses whether integrating symbolic domain knowledge with neural generation improves oracle accuracy and correctness beyond the best purely neural approaches.

The PhD thesis contributes to the state of the art in automatic test oracle generation providing three main results:

C-1 A comparative evaluation of neuro-symbolic and pure neural approaches for generating axiomatic oracles (Chapter 3): The work contributes with TRATTO, a neuro-symbolic approach to automatically generate axiomatic test oracles, and a rigorous comparative evaluation against purely symbolic and purely neural methods for axiomatic oracle generation on benchmark datasets. TRATTO infers executable axiomatic assertions, token-by-token, from source code and documentation. The symbolic module of TRATTO

restricts the search space of the tokens that the neural module use to generate valid oracles, by exploiting the grammar of the programming language, the unit under test, and the context of the unit (its class and the available APIs). The neural module of TRATTO uses transformers fine-tuned for both deciding whether to produce an oracle or not and selecting the next lexical token to incrementally build the oracle from the set of tokens returned by the symbolic module. TRATTO achieves 73% accuracy, 72% precision, 61% F1-score on a ground-truth dataset, outperforming a representative state-of-the-art symbolic approach (Jdoctor: 61%,62%, and 25%) and a strong neural baseline (GPT-4 complemented with few-shot and Chain-of-Thought: 40%,24%, and 37%). It produces $\sim 3\times$ more correct axiomatic oracles than Jdoctor and $\sim 10\times$ fewer false positives than GPT-4. The evaluation validates the research hypothesis, and shows that symbolic integration with neural generation improves oracle correctness. The findings also highlight some challenges in extending these benefits to more complex, open-ended scenarios.

C-2 A large-scale empirical study of vanilla LLMs for concrete test oracles (Chapter 4): The work contributes with an unbiased post-training-cutoff dataset of 13,866 mined from 135 Java projects and an evaluation grid spanning multiple model families, types and sizes, different prompt templates and context configurations. In the experiments, LLMs inferred oracles with average mutation score of 43% — similar to the 45% score of human-designed test oracles. The results indicate that the test prefix and the methods called in the program under test provide sufficient information to generate good oracles, while additional code context does not bring relevant benefits. These findings provide actionable insights into using LLMs for automatic testing and highlight their current limitations in generating complex oracles.

C-3 A comparative evaluation of neuro-symbolic and pure neural approaches for generating concrete oracles (Chapter 5): The work defines TRACTO, a neuro-symbolic concrete test oracle generator, and conducts a systematic comparison involving *Qwen2.5-Coder (32-billion-parameters)*, the best performing model according to the empirical study (**C-2**), and a pure neural model from the same family fine-tuned for the task of concrete test oracle generation. TRACTO achieves 80% compilation rate and 59% test pass rate for successfully generated oracles, outperforming the vanilla baseline (10% and 7%) and the fine-tuned neural model (73% and 51%). The comparison demonstrates that symbolic constraints improve the correctness of successfully generated oracles—TRACTO achieves 7% higher compilation rate and 8% higher test pass rate than the pure neural model—while introducing a trade-off in test case completion. Since the neural component of TRACTO and the fine-tuned neural model share the same underlying neural architecture and training data, the comparison effectively serves as an ablation study isolating the effect of symbolic constraints. The results support the research hypothesis: symbolic integration improves oracle correctness metrics compared to pure neural approaches.

The main results presented in this thesis have been published at international venues, and all tools and datasets are publicly available. We presented TRATTO [51] at the ACM SIGSOFT International Symposium on Software Testing and Analysis. We presented the empirical study on vanilla LLMs at the IEEE/ACM International Conference on Automated Software Engineering [50]. We are currently working on a submission with the results of TRACTO to ACM Transactions for Software Engineering and Methodology. The tools and datasets are available via

the STAR lab website [1], and the author's GitHub [2] profile; replication packages accompany the published papers. The author of the thesis contributed either wholly or significantly to the novel ideas presented in all papers, to the implementation of the tools, and to the experimental evaluation of the results.

1.2 Document Structure

This thesis is organized as follows.

Chapter 2 surveys the state of the art in generating assertion-based test oracles, with emphasis on symbolic- and neural-based techniques, to generate assertion-based oracles. It discusses common threats to validity in neural-based test oracle generation and motivates a neuro-symbolic perspective on test oracle generation by surveying successful neuro-symbolic techniques applied in code-related tasks, such as code generation and test repair. The discussion establishes the conceptual and methodological foundation for the comparative investigation carried out in subsequent chapters.

Chapter 3 presents TRATTO, our neuro-symbolic approach for generating axiomatic test oracles: it details the symbolic search-space restriction (grammar, typing, and scope constraints) phase, the transformer-based token selection process, the datasets and training protocol, baselines and ablation studies, the analysis on correctness with quality metrics (accuracy, precision, recall, and F1-score), the experiments on robustness to documentation variability, and the impact of generated oracles on mutation scores when axiomatic test oracles are integrated with automated test case generators. The results provide the first empirical evidence within this work to test the research hypothesis, highlighting the strengths and limitations of neuro-symbolic generation in the axiomatic domain.

Chapter 4 reports a large-scale empirical study assessing the capabilities of large language models (LLMs) in generating concrete test oracles. It introduces a post-training-cutoff dataset and an evaluation matrix covering multiple model families, sizes, prompt structures, and contextual configurations. The chapter analyzes accuracy, precision, effectiveness, and sensitivity to prompt and model design, deriving guidelines for configuring the neural component of the neuro-symbolic architecture. These findings establish the empirical baseline used in Chapter 5 to test the research hypothesis under controlled comparative conditions.

Chapter 5 introduces TRACTO, a neuro-symbolic approach for generating concrete oracles. TRACTO combines symbolic constraints for syntax and type validation with neural generation for token selection and literal inference. The chapter outlines the system architecture, detailing the grammar-based analysis, identifier resolution mechanisms, validation constraints, and the token-by-token generation workflow. The chapter compares TRACTO against both the best-performing vanilla model identified in Chapter 4 and a pure neural model fine-tuned on the same training data without symbolic constraints, effectively isolating the contribution of the symbolic component through an ablation study. The comparative evaluation measures accuracy, compilation rate, and test pass rate on an unbiased post-cutoff test set. The results demonstrate that TRACTO achieves superior per-oracle correctness compared to both the pure neural model and the vanilla baseline, validating that symbolic integration measurably improves oracle quality. The results support the research hypothesis that symbolic constraints

[1] https://star.inf.usi.ch/#/software-data
[2] https://github.com/darthdaver?tab=repositories

improve oracle correctness, while revealing a trade-off between per-oracle quality and test case oracles coverage.

Chapter 6 summarizes the overall findings, revisits the research hypothesis in light of the comparative evidence, and discusses methodological and practical implications. It outlines limitations of the approach, and discusses future research directions, including strategies for contextually adaptive neuro-symbolic prompting, where contextual information evolves dynamically during oracle synthesis. The chapter concludes by positioning the thesis contributions within the broader landscape of automated test oracle generation and neuro-symbolic software engineering.

Chapter 2

Related Work

This chapter surveys the state of the art in generating test oracles, with emphasis on symbolic- and neural-based techniques to generate assertion-based oracles. The chapter introduces the fundamental concepts of test oracles, including soundness, completeness, correctness, and strength. It examines approaches to automatic test oracle generation, categorizing them into symbolic and neural techniques. It discusses common threats to validity in neural-based test oracle generation. Finally, the chapter further motivates a neuro-symbolic perspective on test oracle generation by surveying successful neuro-symbolic techniques applied in code-related tasks, such as code generation and test repair. The discussion establishes the conceptual and methodological foundation for the comparative investigation carried out in subsequent chapters.

2.1 Test Oracles Fundamentals

In their widely cited survey paper, Barr et al. classify test oracles as *(i) specified test oracles*, *(ii) implicit test oracles*, and *(iii) derived test oracles* [4].

Specified test oracles are oracles generated from formal specifications, model-based specification languages [68, 69], state transition systems [43, 46], assertions and contracts [48, 29], and algebraic specifications [7]. Formal specifications are not always available, and formal specification models rely on abstractions that can lead to the definition of imprecise or infeasible behavior, limiting their applicability in practice.

Implicit test oracles are oracles that leverage general and implicit knowledge to distinguish between correct and incorrect behavior. Unlike specified test oracles, implicit oracles require neither domain knowledge nor a formal specification, and apply to nearly all programs. Classic implicit oracles reveal null-deference and program crashes, among other examples. Implicit test oracles are easy to automate: Fuzzing techniques [6] and test generation tools, such as Randoop [54], successfully exploit implicit oracles to discover anomalies. However, implicit test oracles can detect only a few kinds of failures.

Derived test oracles are oracles inferred from all artifacts but formal models, such as informal documentation and system executions. Relevant derived oracles are regression oracles [76], which check for deviations of behaviors between different versions of a system, and metamorphic relations [11], which check for known relations between mutation of the inputs and corresponding changes of the outputs. Regression oracles cannot reveal faults in the current

version of the code, while the effectiveness of metamorphic oracles is limited to cases where equivalence properties exist and are easy to exploit.

In the absence of formal specifications, implicit and derived test oracles provide a viable approach to test the behaviors of a system. To unleash their full potential, test generation techniques must go beyond implicit, regression, and metamorphic oracles, and shall capture the semantic of the software to generate test oracles that can check the conformance of test execution with the intended program behavior.

Oracles capture the expected behavior of the software system with different degrees of precision. The most typical issue with test oracles is the misclassification of invalid program states as valid (*false negative*), and of valid program states as not valid (*false positive*). Although both reveal inaccuracy in the information captured in the oracles, their impact is different: *False negatives* derive from weak test oracles that can detect only some faulty situations. *False positives* classify valid program states as invalid and denote the inability of the test oracles to properly observe the intended behavior of the system.

Soundness and Completeness

The quality of a test oracle is assessed by two essential properties: *soundness* and *completeness*. These properties are defined with respect to a conceptual ground truth oracle G, which always provides the correct answer [4].

Soundness is a measure of the oracle's integrity. It answers the question: "*When the oracle claims the system is broken, can we trust it?*". A test oracle O is *sound* if every time O provides a positive answer to a proposition w (in the context of software testing it corresponds to an assertion that exposes a bug), this answer is also true according to the ground truth oracle G (in formal terms, $O(w) = 1 \implies G(w) = 1$). A sound test oracle never raises false alarms: whenever it diagnoses an incorrect behavior, the verdict is indeed correct according to the ground truth (the test cases comprising the oracle do not suffer from *false positives*). However, a sound oracle may still miss some faulty behavior, leading to *false negatives*, that is, executions that should be rejected but are instead accepted (the related test cases do not fail when they should).

Completeness is a measure of the oracle's coverage of the fault space. It answers the question: "*If the system is broken, will the oracle notice?*". A test oracle O is *complete* if every time G provides a positive answer on a test activity w, exposing a bug, oracle O returns the same positive verdict (this implication logic translates to $G(w) = 1 \implies O(w) = 1$). A complete oracle never miss a faulty behavior: every actual failure in the system is captured and reported by the oracle. When inserted into test cases, a complete oracle never produces *false negatives*. However, a complete oracle may be overly restrictive: it might reject some behaviors that are actually correct, leading to *false positives* (the related test cases fail when they should not).

While test oracles cannot, in general, be both sound and complete [4], practitioners must use partially correct test oracles. The tension between soundness and completeness represents a fundamental trade-off in test oracle design: oracles prioritizing soundness (conservative about acceptance) reduce false positives at the cost of missing some bugs, while oracles prioritizing completeness (more permissive) detect all faults but accept false positives as the trade-off.

Oracle Correctness and Strength

Beyond soundness and completeness, two additional properties characterize oracle quality in practice: *correctness* and *strength*.

Oracle *correctness* measures whether generated oracles accurately represent intended program behavior. A correct oracle distinguishes faulty executions from correct ones without producing false positives or false negatives [31]. Correctness evaluation requires either comparison against ground truth oracles or execution-based validation that confirms generated assertions pass on the original code. The correctness of a test oracle cannot, in general, be automatically proven in the absence of a ground truth oracle. Even when assuming that the tested code behaves correctly (so that any rejection would constitute a false positive), it remains practically impossible to demonstrate that the oracle never produces false negatives. This limitation stems from the undecidability of the general program correctness problem and highlights the empirical rather than formal character of oracle evaluation.

Oracle *strength* measures the fault detection capability of oracles, that is, their ability to distinguish correct program behavior from faulty behavior [31]. Weak oracles may be syntactically correct but fail to detect many types of faults, while strong oracles effectively expose diverse classes of errors. Oracle strength strongly correlates with mutation testing metrics: oracles that kill many mutants have fault detection capability [2, 38].

The concepts of correctness and strength capture the dual requirements for practical test oracles. Correctness ensures oracles do not interfere with correct program execution (avoiding false positives), while strength ensures oracles effectively detect faults when they occur. These concepts relate to soundness and completeness: a perfectly correct oracle would be sound (no false positives), while a maximally strong oracle would approach completeness in fault detection (detecting all possible faults). However, correctness and strength represent operational, measurable properties rather than theoretical ideals: throughout this thesis, when evaluating oracle correctness, we refer to partial correctness, meaning that the oracle is considered correct to the extent that its observed behavior aligns with expected outcomes under the evaluated conditions.

2.2 Symbolic Approaches for Automatic Test Oracle Generation

Symbolic approaches leverage static and dynamic program analysis and natural language processing (NLP) to understand software semantics and produce semantically relevant oracles in the absence of formal specifications. Symbolic techniques refer to methods that use symbolic values, expressions, and predefined patterns to validate the requirements of software systems. The literature distinguishes between symbolic techniques that capture the semantics of a program through the execution of the source code, and the ones that pursue the same goal through the static analysis of the code and the comprehension of the informal specification accompanied to it.

Invariant mining methods and detectors execute a program on a collection of inputs (test cases) against a collection of potential invariants: Daikon [20] dynamically infers likely invariants from those invariants not violated during the program executions over the inputs. The inferred invariants capture program behaviors, and thus can be used to check program correctness. Dysy [16] combines symbolic program execution with dynamic execution of the test suite to improve the quality of the inferred invariants and reduce the number of test cases required to disqualify the irrelevant ones. The accuracy of the invariants inferred with these methods depends on both the quality and completeness of the test cases and the collection of potential invariants provided. Evospex [49] combines observed executions with mutations to generate samples of both valid and likely invalid program states and applies a genetic algorithm to in-

fer invariants for method postconditions. GAssert [63] proposes a technique to automatically improve inferred assertion oracles, reducing false positives and negatives with an evolutionary algorithm. These approaches derive specifications and test oracles relying on the execution of the current version of a program, therefore they generate regression oracles that cannot detect if the bug is already present in the program.

Text-driven specification mining methods exploit natural language processing, pattern, semantic, and syntax matching to generate test oracles from code comments and text documentation. ALICS [55] mines code contracts from API documentation. The study considers code contracts in the form of preconditions and postconditions. ALICS uses NLP-based pattern matching with an application domain-specific dictionary, thus it can hardly generalize and generate contracts for API documentations with unseen structures. @tComment [61] defines natural language patterns and heuristics to extract null value exceptions from Javadoc comments. It cannot generalize to other properties or exception types.

Toradocu [27] applies lexical, similarity matching and natural language patterns to the semi-structured portion of Java code documentation (JavaDoc), that is, comments labeled with *"@"* tags. Toradocu generates oracles from *@throws* tags, which describe exceptional conditions, and generates exception oracles. Toradocu can only generate oracles for exceptional behavior. Jdoctor [8] extends Toradocu, by considering also *@param* and *@return* tags, and can generate both exception and assertion oracles, in the form of preconditions and postconditions. Memo [9] processes the unstructured part of JavaDoc comments to identify metamorphic relations expressed in the text and translate them into executable assertion oracles.

These methods can precisely determine oracles when code comments fit the defined patterns, but do not generalize when comments fall outside these patterns. Moreover, these approaches are not applicable when code comments are unavailable.

2.3 Neural Approaches for Automatic Test Oracle Generation

Neural oracle generation techniques leverage deep learning and large language models (LLMs) to generate test oracles by learning from a large corpora of code and test data [57]. These approaches promise better generalization beyond fixed patterns but introduce challenges related to correctness, false positives, and oracle strength [32].

Recurrent Neural Networks for Oracle Generation

ATLAS [72] pioneered the application of recurrent neural networks to test oracle generation. The system takes as input a test prefix and the method under test, and learns to generate assertion oracles by identifying patterns in developer-written tests. ATLAS trains a sequence-to-sequence LSTM [30] model on pairs of test contexts and their corresponding assertions, automatically extracted from large collections of open-source projects.

ATLAS demonstrates that neural models can autonomously learn to generate assertions without relying on manually defined rules or patterns, marking a fundamental shift from traditional symbolic approaches. The model learns to select appropriate assertion types (such as '`assertEquals`', '`assertTrue`', '`assertNull`, etc.) and generates assertion arguments based on variables and method calls available in the test prefix. ATLAS relies only on the source code, does not have any knowledge of the code documentation; it targets only assertion oracles, and cannot generate oracles for exceptional behavior.

Transformer-based Oracle Generation

Tufano et al.'s approaches [66, 67] outperform ATLAS by replacing the recurrent neural network with transformer models pre-trained on natural language and code. AthenaTest [65] is the first noteworthy approach that generates test cases that include both test prefixes and oracles, considering both the unit implementation and its context, such as the surrounding class and the method signature. In AthenaTest, the information provided by the code documentation is still not considered. TOGA [19] is a major advance in neural oracle generation, utilizing CodeBERT transformers fine-tuned for both exception and assertion oracle generation. TOGA introduces a two-stage pipeline that addresses the oracle generation problem more comprehensively than ATLAS and AthenaTest. In the first stage, an exception classifier determines whether a test should expect an exception or an assertion oracle. In the second stage, separate models handle exception oracle generation (predicting the expected exception type) and assertion oracle generation.

TOGA extracts candidate assertions from the test prefix using rule-based heuristics that identify all possible assertions on variables and method return values. A neural model then evaluates these candidates and selects the most likely correct assertion. This hybrid approach combines symbolic candidate extraction with neural ranking, reducing the generation space while leveraging learned patterns to select semantically meaningful oracles.

TOGA demonstrates superior performance that ATLAS in detecting real faults, identifying 57 bugs in the Defects4J benchmark including 30 unique bugs not detected by other oracle generation methods. This strong empirical result suggests that neural oracles can provide practical value beyond traditional symbolic approaches. However, Hossain et al.'s large-scale evaluation [32] reveals significant limitations of TOGA. According to Hossain et al.'s evaluation on 25 real-world Java projects with 223,557 test cases, TOGA exhibits high false positive rates, 18% on assertion oracles and 81% on exception oracles, and fails to generate any assertion for 62% of inputs requiring assertions, and among generated assertions, 47% are false positives. Hossain et al.'s mutation testing study provides additional concerns: TOGA assertions increase fault detection by only 0.3% relative to EvoSuite-generated oracles. These findings indicate that while TOGA generates assertions that occasionally detect real bugs, the generated oracles are often weak and do not substantially improve test suite strength.

Large Language Model-based Oracle Generation

Recent work explores leveraging state-of-the-art LLMs for test oracle generation, motivated by their impressive performance on diverse code-related tasks. In [42], Konstantinou et al. investigate whether LLM-generated oracles capture actual program behavior or expected behavior, a crucial distinction for oracle utility . The study evaluates GPT-3.5 Turbo on test oracle classification and generation tasks using prompts with varying amounts of context.

The findings reveal important limitations in the use of large language models (LLMs) for test oracle generation. Their empirical results show that LLMs tend to capture the actual program behavior rather than the expected behavior, achieving less than 50% accuracy in correctly classifying externally provided assertions, and their performance degrades further when the code under test is buggy. This bias implies that current LLMs are better suited for regression testing—checking that existing behavior has not changed—than for detecting logic or specification faults that require inferring developer intent. On the generation task, the study reports a more positive outcome: when LLMs are asked to produce assertions, about 60% of generated assertions are directly valid (i.e., they compile and pass), and in roughly 90% of the test prefixes the

model manages to produce at least one correct assertion when instructed to generate five different test oracles. Furthermore, the authors show that LLM performance depends on the quality of program context: when tests and variables use meaningful, developer-like names, accuracy improves by up to 16.1% compared to Evosuite-style names. Finally, LLM-generated oracles achieved higher mutation scores (up to 2% better) than Evosuite oracles, suggesting that, despite their tendency to mirror actual behavior, they are valuable for test-suite augmentation and can strengthen automated testing pipelines.

TOGLL [31] builds on the foundation of LLM-based test oracle generation by fine-tuning seven distinct code-specific language models, using six prompt strategies on the SF110 dataset presented in [24], which features 110 diverse Java projects. The study demonstrates that smaller dedicated models such as CodeParrot-110M, when coupled with rich context containing signatures, documentation, and code bodies, can match or outperform larger LLMs for oracle generation. On OracleEval25, a dataset of 25 unseen projects presented in [32], TOGLL produces up to 3.8 times more correct assertion oracles and 4.9 times more correct exception oracles compared to prior neural oracle synthesis approaches. Noteworthy, only 9.5% of TOGLL's assertion outputs exactly match the ground truth oracles, confirming broader coverage and strong generalization beyond training data. In bug detection experiments using Defects4J [40], TOGLL detects ten times more unique mutants and 106% more real bugs than the best previous baseline, setting a new performance standard for LLM-driven test oracle generation. This work shows that providing detailed code and documentation context enables LLMs to synthesize oracles that are correct, diverse, and strong, revealing that fine-tuned code models can surpass general-purpose approaches both in accuracy and practical bug-finding impact.

Recently, AugmenTest [41] presents an alternative approach that leverages LLMs to infer correct test oracles based on available documentation of the software under test. Unlike most existing methods that rely on code, AugmenTest utilizes the semantic capabilities of LLMs to infer the intended behavior of a method from documentation and developer comments, without examining the code implementation. The approach includes four variants: Simple Prompt, Extended Prompt, RAG with a Generic Prompt, and RAG with Simple Prompt, each offering different levels of contextual information. AugmenTest employs a flexible, model-agnostic framework that can be integrated with any LLM, supporting both closed-source APIs and local quantized models. The evaluation of AugmenTest employed a dataset of 203 test cases drawn from 142 Java classes, where mutation-based fault injection was used to create controlled behavioral deviations and ensure the tests captured semantically meaningful changes. The results demonstrate that, in the most conservative scenario, AugmenTest's Extended Prompt achieved a 30% success rate for generating correct assertions, significantly outperforming the state-of-the-art TOGA approach which achieved 8.2%. Interestingly, the RAG-based variants (which rely on retrieval-augmented context) underperformed, suggesting that further refinement is needed for effective integration of structured data. In mutation testing experiments, The Simple Prompt achieved a comparable success rate of 29.1%, whereas RAG-based variants performed worse (18.2%). Failure rates ranged from 30.9% for the Simple Prompt to 41.8% for the Extended Prompt, highlighting the ongoing difficulty of assertion generation. None of the approaches were able to infer exception oracles. These results demonstrate substantial improvement in assertion inference over neural baselines, but highlight ongoing limitations for exception handling and consistency in oracle generation.

Prompt Engineering and Context Effects

Prompt design critically shapes LLM oracle generation, with the richness and relevance of contextual information directly affecting oracle correctness and strength. Hossain et al. [31] systematically evaluated six prompt variations with increasing contextual information on seven code large language models (PolyCoder-400M/2.7B, CodeGen-350M/2B/6B, CodeParrot-110M, InCoder-1.3B/6.7B). Their results prove substantial performance differences across prompt designs: test prefix alone achieves 55.4% average exact match accuracy, adding method signatures improves this to 75%, and including full method implementations yields the highest accuracy of 77%. The study reveals that smaller fine-tuned models (110M-400M parameters) meet or exceed the performance of larger models when provided with appropriate prompts, suggesting that prompt design may be more critical than model scale for oracle generation tasks.

In a follow-up paper [33], Hossain conducted a comprehensive investigation of documentation's role in test oracle generation, systematically evaluating three prompt pairs that progressively incorporate Javadoc comments alongside test prefix, method signature, and method implementation. Their key finding challenges the assumption that method implementation is necessary for effective oracle generation: using Javadoc comments alone (with test prefix) achieves 78.11% accuracy on average across the same seven code LLMs presented in [31], approaching the 80.83% achieved when using full method implementations along with the relative documentation. This result is particularly significant because using documentation avoids the risk of learning buggy behavior from defective implementations. The study demonstrates that including Javadoc comments improves test oracle generation performance by 10-20 percentage points when the base prompt contains minimal information. Through ablation studies, removing different Javadoc components, the authors identify that method descriptions and return tags provide the greatest value—removing descriptions reduces accuracy by 10 percentage points (from 78.37% to 68.12%), while removing return tags reduces accuracy by 5.45 percentage points. In contrast, removing parameter descriptions, exception specifications, or cross-references causes accuracy drops of less than 1 percentage point. These findings indicate that when LLM token limits constrain prompt length, prioritizing descriptions and return value specifications maximizes oracle quality while other documentation elements can be omitted without significant performance degradation. The study also explores using GPT-generated documentation to supplement missing human-written comments, showing 10 percentage point improvements over using no documentation, suggesting that automated documentation generation may partially address quality variation in real-world code-bases.

The prompt design investigations conducted by Konstantinou et al.[42] and Khandaker et al.[41], discussed earlier in terms of overall performance, also provide specific insights into prompt structure effects. Konstantinou et al. demonstrate that method signatures improve accuracy by 20 percentage points over test prefix alone, supporting the findings of Hossain but using a pre-trained LLM (GPT-3.5 Turbo versus fine-tuned code models) and evaluation dataset (GitBug-Java versus SF110). This consistency across different experimental settings strengthens confidence in the importance of method signatures for oracle generation. Their investigation of naming conventions reveals that meaningful test and variable names improve accuracy by up to 16.10 percentage points compared to automatically generated generic names, highlighting that prompt effectiveness depends not only on what information is included but also on how that information is presented. Khandaker et al. compare Simple Prompts (test prefix + focal method signature) against Extended Prompts incorporating class-level information (constructors, methods, and fields), finding modest improvements of approximately 1 percentage point.

Their evaluation of RAG-based prompts, which retrieve similar test cases to provide few-shot examples, reveals unexpected under-performance (18.2% success rate versus 30% for Extended Prompt), suggesting that retrieved examples may introduce noise rather than helpful guidance when examples are not sufficiently similar to the target test case.

These collective findings emphasize that effective LLM-based oracle generation requires careful prompt engineering to leverage available context, but they also reveal fundamental tensions in prompt design. Optimal prompts require substantial context (full method code or comprehensive documentation), yet longer contexts increase computational costs and may exceed model context windows for large methods. Including method implementations risks learning buggy behavior when code contains defects, while excluding them may limit semantic understanding. Documentation provides a middle ground, offering semantic information without implementation-level details, but documentation quality varies significantly in practice. Different models exhibit varying sensitivity to prompt variations, with some models showing greater improvements from documentation inclusion than others. The substantial impact of naming conventions indicates that oracle generation approaches must be evaluated on both developer-written tests with meaningful names and automatically generated tests with generic identifiers to assess real-world applicability across different testing scenarios. These insights inform the design of prompts for concrete oracle generation and highlight the importance of balancing context richness with practical constraints.

Limitations of Neural Approaches.

Current neural-based approaches demonstrate noticeable capability to generate oracles without executing the code under test and also in the absence of code documentation, overcoming the limits of symbolic-based approaches in terms of applicability. However, this flexibility comes with a cost in terms of accuracy, since these methods struggle to generate accurate oracles due to the large space of possible assertions. Hossain et al.'s [32] evaluation of the current neural-based test oracle generation approaches highlights that the inferred assertions exhibit high false positive rates that threaten their practical usefulness.

2.4 Neuro-Symbolic Approaches for Automatic Test Oracle Generation

The initial results and the successful applications of neuro-symbolic techniques in related domains, such as code generation and software testing more broadly, incentive the investigation of neuro-symbolic techniques for generation test oracles.

Neuro-Symbolic Code Generation

Recent work in code generation demonstrates that combining neural models with symbolic constraints significantly improves both correctness and efficiency. Bunel et al. [10] demonstrate similar benefits for program synthesis by leveraging domain-specific language (DSL) grammars combined with reinforcement learning. Bunel e al.'s approach tackles two key limitations of purely neural synthesis: (i) program aliasing, where multiple programs can satisfy the same specification but training optimizes only for a single reference implementation, and (ii) the lack of syntactic guarantees, by performing reinforcement learning with an objective that explicitly

maximizes the likelihood of generating semantically correct programs, and introducing training procedures that enforce syntactic correctness via grammar constraints.

Fu et al. [25] explore constrained decoding for generating secure code, formulating the problem as generating programs that satisfy both correctness and security constraints. Fu et al. introduce constraint specifications based on secure coding practices and vulnerability types, and propose constrained beam sampling and gradient-based non-auto regressive decoding techniques to enforce these constraints during generation. The evaluation on SecurityEval and SVEN benchmarks shows that constrained decoding reduces security vulnerabilities up to 17% compared to unconstrained generation while maintaining comparable functional correctness. The approach shows that interpreting the decoding process rather than treating it as a black box uncovers new opportunities to enhance neural code generation through symbolic constraints, and achieves stronger security and correctness guarantees than purely neural methods.

Mündler et al.[52] introduce type-constrained decoding for code generation, where LLMs are guided by type system rules during token generation. Their approach constructs prefix automata that maintain type-relevant context and enforce type correctness at each generation step, ensuring that partial programs can be completed into well-typed final programs. On TypeScript code synthesis, translation, and repair tasks across HumanEval and MBPP benchmarks, type-constrained decoding reduces compilation errors by more than 50% compared to unconstrained generation and increases functional correctness by 3.5% to 5.5%. Notably, syntax-only constraints account for 6% of compilation error reductions, while type constraints address 94% of errors, demonstrating that semantic constraints provide substantially greater value than syntactic constraints alone.

The approach employs a type search algorithm that determines whether partial expressions can inhabit required types by exploring sequences of operators and member accesses. This search over a type graph (where nodes represent types and edges represent well-typed operations) enables the system to prune invalid generation paths while maintaining a prefix property that guarantees eventual completion to well-typed code. The technique proves broadly applicable across different LLM sizes (2B-34B parameters) and model families, with even small models benefiting significantly from constraints.

Neuro-Symbolic Software Testing

Some studies indicate the suitability of neuro-symbolic approaches in software testing. Chen et al.'s [12] FlakyDoctor a neuro-symbolic technique combines LLMs with program analysis to repair test flakiness. The approach achieves 57% success rate for order-dependent flakiness and 59% for implementation-dependent flakiness, outperforming purely symbolic approaches by up to 17%. Chen et al.'s ablation study reveals that the symbolic (non-LLM) components contribute up to 31% of overall performance, demonstrating that while part of the FlakyDoctor capability derives from using LLMs, they are not good enough to repair flaky tests in real-world projects alone. This finding emphasizes that symbolic reasoning provides complementary value that LLMs cannot achieve independently.

Implications for Test Oracle Generation

These successful applications of neuro-symbolic techniques in related domains provide strong motivation for exploring neuro-symbolic approaches to test oracle generation. Several parallels emerge between these domains and oracle generation:

1. Compilability and correctness are critical in both code and oracle generation. Type-constrained decoding demonstrates that enforcing formal language rules (type systems, grammars) during neural generation substantially improves output quality. Test oracles similarly require syntactic correctness (valid Java/Python syntax), type correctness (assertions on compatible types), and semantic correctness (checking meaningful properties). The observation that type constraints address 94% of compilation errors while syntax constraints address only 6% in [52] suggests that oracle generation would similarly benefit more from semantic constraints (valid assertion types, type-compatible arguments) than from purely syntactic constraints.

2. Constrained token-by-token generation enables tight integration of symbolic reasoning. Neuro-symbolic approaches can constrain each generation step to prevent the model from entering invalid states, rather than generating complete oracles in a single forward pass and then validating them. This incremental constraint application provides stronger correctness guarantees and reduces the search space the neural model must explore, improving both efficiency and accuracy.

3. Combining neural and symbolic strengths addresses complementary limitations. Neural models excel at learning patterns from large code corpora and generalizing across diverse coding styles but struggle with formal correctness guarantees and rare patterns. Symbolic approaches provide correctness guarantees and handle formal rules precisely but lack generalization beyond predefined patterns. Neuro-symbolic integration leverages neural generalization while maintaining symbolic correctness, potentially achieving better performance than either paradigm alone.

4. Empirical evidence from related domains shows substantial improvements. Code generation guided by type-constraints achieves more than 50% error reduction in [52], grammar-constrained program synthesis improves accuracy with limited training data in [10], constrained decoding for secure code reduces vulnerabilities up to 17 in [25], and neuro-symbolic test repair outperforms pure symbolic and pure neural baselines in [12]. These consistent improvements across multiple software engineering tasks suggest that neuro-symbolic approaches represent a promising direction for test oracle generation.

The current lack of neuro-symbolic work specifically on test oracle generation represents a research opportunity. While symbolic approaches like Jdoctor [8] achieve high precision and recall, but fail to generate oracles when the specification falls outside the defined set of domain-specific rules, and neural approaches like TOGLL achieve broader coverage, but suffer from false positives (25% for assertions, 7% for exceptions), a neuro-symbolic approach can combine symbolic precision with neural coverage. The successful integration of type systems, grammars, and security constraints in code generation demonstrates the feasibility of incorporating domain-specific formal rules into neural generation. Applying similar techniques to test oracle generation (constraining assertion types, argument types, variable scopes, and semantic properties) can improve both correctness and strength of generated oracles.

Chapter 3

Tratto: a Neuro-Symbolic Approach for Generating Axiomatic Oracles

In this chapter, we address the problem of automatically deriving axiomatic test oracles and we present TRATTO, a neuro-symbolic approach to derive axiomatic oracles from commonly available software artifacts such as source code and documentation. TRATTO reformulates the oracle problem into a token generation problem. The symbolic module integrated in TRATTO restricts the space of the candidate tokens that can be used to iteratively generate an oracle, always compilable by construction. The neural module guides the generation of oracles towards an optimal solution, that is, an oracle that captures the expected behavior of the unit under test. The core contributions of this chapter are the main content of a paper that we presented in the technical track of the 2025 International Symposium on Software Testing and Analysis ([51]).

Axiomatic oracles can be encoded as *preconditions* and *regular* or *exceptional postconditions* on a unit under test. A *precondition* specifies the requirements that the inputs must satisfy for the unit to operate correctly, ruling out invalid program inputs that contribute to false positives. *Regular postconditions* describe the expected properties of the outputs or final state on normal execution. *Exceptional postconditions* capture conditions under which the method must throw exceptions, effectively describing when an exception should occur. State-of-the-art approaches to generate test oracles produce concrete oracles, that is, assertions on concrete inputs. For instance, a concrete oracle might state that `Math.abs(-5)` returns `5` for one single input. Axiomatic oracle generalizes the property for all inputs, e.g. asserting that `Math.abs(x)` $\geq$ `0` and (`Math.abs(x) = x` || `Math.abs(x) = -x`) for any `x` (an invariant property of the abs function). Similarly, instead of individually checking that the rest of the division between `7` and `3` is always `1` , an axiomatic oracle infers that for any positive integers `a` and `b`, the remainder `a % b` is always `0` $\leq$ `a % b` $<$ `b`. In essence, axiomatic oracles act like the design-by-contract specifications of a unit, defining requirements on the input and output values, similarly to the contracts in Eiffel [48] or the invariants mined by tools like Daikon [20]. Such general oracles are more powerful than example-based assertions because they apply to all test executions of a unit, albeit at the cost of reduced point-wise precision.

TRATTO (TRAnsformer-based Token-by-Token Oracle generation) generates axiomatic or-

acles in the form of executable assertions, from source code and documentation. Although the theoretical foundations of TRATTO are, in principle, language-independent, its current implementation expresses axiomatic oracles as boolean expression in Java. TRATTO derives the assertions from Java source code and the corresponding Javadoc documentation. Specifically, TRATTO infers preconditions from `@param` tags, regular postconditions from `@return` tags, and exceptional postconditions from `@throws` tags. In addition, TRATTO analyses the free-text portions of Javadoc comments, as well as method signatures and available implementations, to enhance the informational context for inference.

TRATTO reformulates the oracle generation problem as a token-by-token code synthesis task. An axiomatic oracle consists of lexical tokens. For instance, the axiomatic oracle "`result > 0;`" is composed of four tokens: '`result`', '`>`', '`0`', and '`;`'. At the lexical level, a token represents an atomic element of the programming language syntax, an indivisible unit that cannot be further decomposed without losing its semantic meaning. This notion of lexical token differs from that used in the context of deep learning transformers, where a token denotes an atomic constituent in the model's vocabulary used to encode natural language into machine-interpretable numerical identifiers (token IDs). Consequently, a single lexical token, such as a variable name '`someVar`', may be mapped to multiple transformer tokens (e.g., '`some`' and '`Var`').

Rather than producing a complete assertion in a single step - as done in prior work - TRATTO incrementally constructs the oracle one lexical token at a time. This iterative formulation enables a tight integration between a symbolic module, which enforces syntactic and semantic correctness at each generation step, and a neural module, which leverages data-driven learning to progressively guide the construction process toward a semantically sound and contextually relevant oracle.

At each iteration of token generation, the symbolic component of TRATTO narrows the search space for the next possible tokens through two complementary mechanisms. The first restriction relies on the programming language grammar and the portion of the oracle already generated. For example, a boolean expression cannot serve as the argument of the '>' operator. The second restriction depends on the symbols currently in scope, which include method parameters and the return value (for postconditions), as well as the fields and methods belonging to the current class or accessible through it.

The neural component of TRATTO selects a token from the set of candidates. It leverages pre-trained transformer models fine-tuned for the task of predicting candidate tokens based on (i) the portion of the oracle generated so far, (ii) the unit under test (including its source code and documentation) and (iii) the broader unit context, such as information about available APIs.

The neural component of TRATTO features a multitask model fine-tuned on a dataset represented in two distinct forms, corresponding to two distinct learning tasks: a dataset of complete oracles and a dataset of tokens extracted from the decomposition of the original oracles. This dataset significantly extends the one provided in the replication package by Blasi et al. [8]. We enhanced the initial data by (i) correcting semantically invalid oracles, (ii) adding oracles inferred from source code and documentation, (iii) incorporating oracles from other publicly available Java projects, and (iv) automatically generating semantically equivalent Javadoc comments that refer to the same oracles. The resulting dataset consists of 34,249 oracle samples and a total of 188,900 tokens derived from them.

We performed a preliminary evaluation aimed to select the most suitable base model for the neural component. We assessed the axiomatic oracle generation capabilities of three leading code generation models at the time — Code Gemma (Google) [62], StarCoder2 (BigCode) [45],

and Code Llama (Meta) [59] — each featuring seven billion trainable parameters. This experiment focused on measuring their accuracy in predicting the next oracle token. Code Llama demonstrated superior performance, achieving 91% accuracy, motivating its selection for fine-tuning and subsequent integration as the neural backbone of TRATTO.

We conducted two ablation studies to isolate the contributions of TRATTO 's symbolic and multitask components. Removing the symbolic module led to a 6-point drop in accuracy, while replacing the multitask model with two independent ones reduced accuracy by a further 3 points. These results confirmed that both the symbolic layer and the multitask learning design are instrumental to oracle accuracy and consistency. We compared TRATTO against two representative state-of-the-art techniques for the generation of axiomatic test oracles: the symbolic approach Jdoctor [8] and the neural GPT-4 model [53]. Evaluations over a curated ground-truth dataset showed that TRATTO substantially outperforms both baselines, reaching 73% *accuracy*, 72% *precision*, and 61% *F1-score* against 61%, 62%, and 25% for Jdoctor, and 40%, 24%, and 37% for GPT-4. GPT-4 achieves a higher *recall* (89%) than both TRATTO (52%) and Jdoctor (16%), however, TRATTO attains an excellent balance between generating oracles (3× more than Jdoctor) while incurring in few false positives (10× less than GPT4).

We conducted additional experiments to investigate the robustness of TRATTO, Jdoctor, and GPT-4 in generating correct oracles when varying the documentation of the tested units (for instance, rephrasing it with equivalent alternatives, or adding typos within the text). TRATTO generated 195 correct and compilable oracles out of 220, closely matching GPT-4 (208) and significantly exceeding Jdoctor (49).

We evaluated the effectiveness of the oracles generated by TRATTO and Jdoctor in improving the mutation scores of test suites automatically produced using EvoSuite [23]. The high proportion of non-compilable oracles generated by GPT-4 (552 out of 1,213), forced its exclusion from the analysis. Among six projects, TRATTO increased the mutation score for 5 test suites containing implicit oracles, and for 5 test suites featuring both implicit and regression oracles. In contrast, Jdoctor improved 4 test suites with implicit oracles but showed no gains with the combined oracle suites.

The remainder of this chapter is organized as follows.

Section 3.1 discusses the limitations of the state-of-the-art approaches to generate axiomatic oracles and provides a motivating example for the work.

Section 3.2 presents the architecture and workflow of the novel approach implemented in TRATTO that iteratively generates test oracles, token-by-token, by combining a symbolic with a neural approach to steer the generation of tokens toward valid oracles, thus reducing the impact of false positives of purely neural approaches.

Section 3.3 details the symbolic module, including the custom grammar and context-based rules.

Section 3.4 describes the neural module, including the multitask learning setup for oracle classification and token generation.

Section 3.5 outlines the procedure for building a collection of comprehensive datasets of oracles and tokens that can be reused for training future models for generating oracles.

Section 3.6 reports the results of the experimental studies conducted to (i) compare the performance of different code models for the tasks of oracle evaluation and token selection; (ii) highlight the contributions of TRATTO 's components to its overall performance; and (iii) compare TRATTO with state-of-the-art approaches for oracle generation, showing its superior performance in terms of correctness, robustness and applicability to enhancing automatically

generated test suites.

Section 3.7 discusses the internal and external threats to validity that may affect the credibility and generalizability of the results, and outlines the measures adopted during the study to mitigate these risks.

3.1 Motivating Example

```
1  /**
2   * Sets the item label generator for a series and
3   * sends a {@link RendererChangeEvent} to all
4   * registered listeners.
5   *
6   * @param series  the series index (zero based).
7   * @param generator  the generator
8   * (<code>null</code> permitted).
9   *
10  * @see #getSeriesItemLabelGenerator(int)
11  */
12 public void setSeriesItemLabelGenerator(int series,
13         CategoryItemLabelGenerator generator) {
14     setSeriesItemLabelGenerator(series, generator, true);
15 }
```

Listing 3.1. JFreeChart `AbstractCategoryItemRenderer#setSeriesItemLabelGenerator`.

Listing 3.1 illustrates the limitations of existing state-of-the-art oracle generation approaches and motivates the design of TRATTO. The Javadoc specification describes a precondition in natural language: *"the series index (zero based)"* (line 6). If a test generator produces a unit test case with a negative `series` as argument, the test crashes: The precondition indicates that the test case is invalid, and executing the test may produce a false positive.

The state-of-the-art oracle generators fail to produce this precondition. TOGA [19], a neural approach for generating test assertions, does not generate preconditions at all. Jdoctor [8], a symbolic approach based on predefined patterns and rules, cannot match this comment format and thus omits the precondition. EvoSuite [23] a search-based test generator, produces only regression test oracles without preconditions. GPT-4 [53], a large language model trained on natural language and code, correctly generates the precondition "`series >= 0`", but also produces incorrect, useless, or non-compilable oracles such as "`generator == null || generator is a valid CategoryItemLabelGenerator instance`" and "`generator == null || generator != null`".

TRATTO successfully derives the precondition "`series >= 0`", preventing the generation of invalid test cases and false positives, without producing additional wrong oracles. The neural module of TRATTO generalizes beyond Jdoctor's fixed set of rules and patterns to previously unseen oracles, while the symbolic module constrains the output space to syntactically and semantically valid, compilable oracles, avoiding the spurious generations observed with GPT-4.

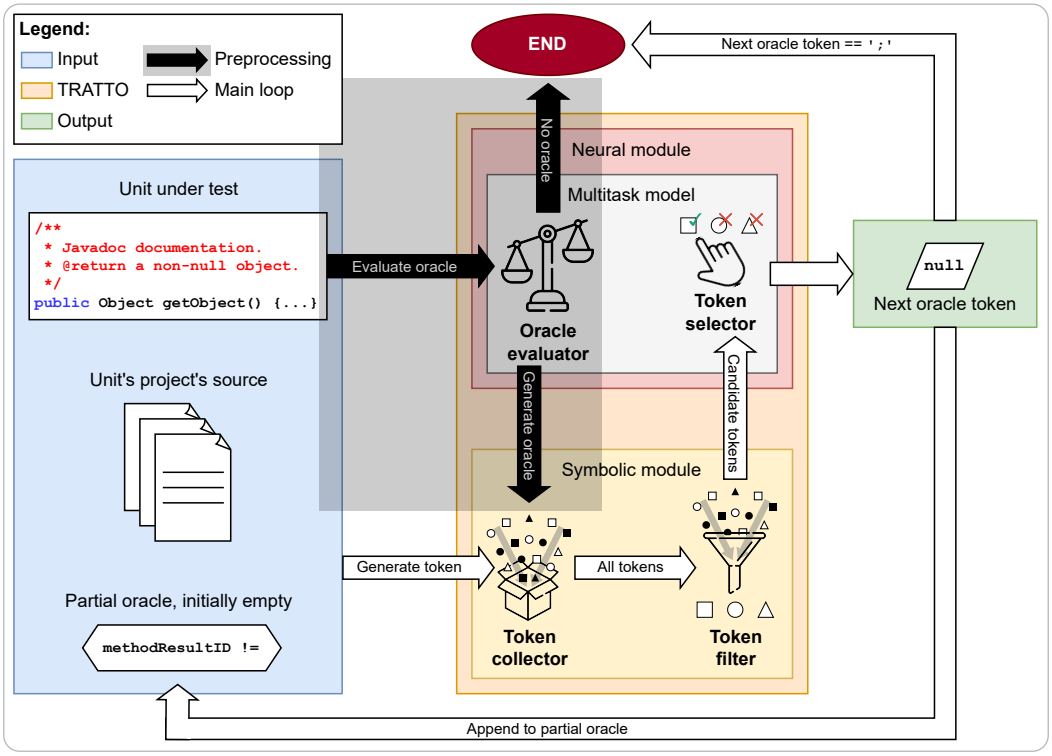

Figure 3.1. Workflow of Tratto.

3.2 Architecture

TRATTO defines a neuro-symbolic approach that incrementally generates test oracles, token-by-token, by integrating a symbolic and a neural module in a tightly coordinated workflow (Figure 3.1). The symbolic module consists of a *Token collector* and a *Token filter*, which, respectively, gather the set of tokens that can form an oracle and restrict them to those that are legal at each generative iteration, ensuring syntactic correctness by construction. The neural module comprises a multitask model that operates in two modes: as an *Oracle evaluator*, it determines whether an oracle should be produced for a given unit under test; as a *Token selector*, it chooses the next token to extend the current partial oracle from among those allowed by the symbolic module.

TRATTO's oracle generation process begins with a pre-processing stage (black arrows in the figure), where the *Oracle evaluator* leverages *documentation* and code of the *unit under test* to decide whether to initiate the oracle generation. If an oracle is required TRATTO triggers the token-generation workflow (white arrows) and proceeds to generate each subsequent token based on the *unit under test*, the associated *project's sources*, and the current state of the *partial oracle*, starting from an empty expression and terminating once the neural module selects a semicolon (';') as the next token. Throughout this iterative process, the symbolic and neural modules cooperates to incrementally generate the tokens that comprise the partial oracle, until they produce a complete oracle: the *Token collector* retrieves all the possible tokens potentially usable to build an oracle (such as symbols, keywords, fields, parameters, or accessible methods

relevant to the context), while the *Token filter* constrains this set to those tokens that maintain oracle validity over the generation (e.g., preventing syntactically incorrect constructions like applying the '`instanceof`' operator to primitives). From the filtered candidate tokens, the neural *Token selector* predicts which token should be appended to the partial oracle.

The token-by-token generation strategy yields significant benefits over traditional methods. Unlike purely symbolic and neural approaches such as Jdoctor and TOGA, which synthesize entire oracles in a single step:

1. The grammar-driven symbolic component implemented in TRATTO: (i) guarantees the generation of oracles compilable by constructor, while neural-based approaches may infer oracles that do not compile [66, 53]; (ii) ensures greater expressiveness in the generation of oracles, in contrast to neural approaches that restrict generation to oracles aligned with a fixed pool of candidate assertions [19]; (iii) automates the extraction of contextual information that enhances the neural component's capability to derive semantically relevant oracles that would otherwise remain inaccessible to purely symbolic or neural approaches without manual intervention or manually specified contextual data.

2. The neural module training at the token level greatly expands the available dataset size, since each oracle comprises multiple lexical tokens. This richer supervision allows the model to learn intricate patterns and associations between natural language and code, overcoming the data limitations typical of approaches reliant solely on pattern-, lexical-, or semantic-matching.

3.3 Symbolic Module

The symbolic module in TRATTO restricts the search space of candidate test oracles by guaranteeing that only syntactically and semantically well-formed, compilable assertions are generated for any given unit under test. The *Token collector* collects all possible tokens that could be used to form an oracle. The *Token filter* discards the illegal tokens (syntactically and semantically invalid tokens that would make the oracle non-compilable) with a grammar-based approach.

3.3.1 Token Collector

The *Token collector* gathers all the possible tokens that serve as the fundamental full set of atomic elements to construct a candidate oracle, given the specific unit under test. During the initial stage of the token-generation process, when the partial oracle is empty, the *Token collector* aggregates three primary categories of *generic tokens*: (i) common tokens which could be part of any oracle, such as operators, keywords, and common constants like `0` and `1`; (ii) tokens extracted from the project under test, including classes and their respective fields and methods (e.g., `CollectionUtils.isEmpty()`); and (iii) tokens extracted from the method under test, including its parameters (if any) as well as fields and methods callable upon those parameters, containing class, and return value of the method (e.g., `this.`**`contains`**`(o)`).

As token generation proceeds, the *Token collector* dynamically refines the set of available tokens to reflect the evolving context of the partial oracle. At each subsequent iteration, the *Token collector* may augment the set of tokens with *specific* tokens that may occur when the last token of the partial oracle is a period that follows an expression that is a class of the project, `this`, `methodResultID` (which identifies the return value of the method under test), or a method

parameter. For instance the *Token collector* adds the tokens '`hasNext`' and '`next`' to the set of tokens when the partial oracle ends with "`this.iterator().`".

3.3.2 Token Filter

The *Token filter* discards any token that would result in syntactic or semantic invalidity if chosen as the next element of the given *partial oracle*. The *Token filter* identifies the tokens to be discarded using a grammar-based pruning approach designed to enforce language rules and compilation safety. Specifically, the *Token filter* (i) discards the tokens that would violate syntactic constraints, such as the boolean value '`true`' that cannot follow '`arg1 >`' in a relational comparison, and (ii) excludes the tokens that would lead to compilation errors given Java's type system and semantics, such as the '`>`' operator that can only follow expressions evaluating to numeric types.

TRATTO enforces also context restrictions to further refine the tokens selection. These rules account for project- and unit-specific semantics that may not be fully captured by the grammar alone. For example, even though the grammar allows a return variable name '`methodResultID`' collecting the result returned by the unit under test, TRATTO discards the token if the method under test is void. The system currently implements 27 documented context restrictions.

3.3.3 Context Restrictions

A grammar alone is insufficient to generate a valid test oracle, as it lacks awareness of the underlying semantics of the specific tokens composing it. For instance, according to the grammar implemented in TRATTO, the token `instanceof` is syntactically valid after the partial oracle `methodResultID`. However, if the type of `methodResultID` (i.e., the return type of the method under test) is a primitive, the resulting expression would not compile.

We refer to such cases as context restrictions, since they constrain token validity based on the semantic and structural context of the oracle under construction. These restrictions extend beyond the grammatical level by incorporating project- and unit-specific semantics to prevent the generation of syntactically correct but semantically invalid oracles.

Currently, TRATTO implements 27 documented context restrictions, each designed to ensure that the generated oracles remain both compilable and meaningful in the context of the method under test. A detailed description of these restrictions is provided in the replication package of the work [3].

3.4 Neural Module

The neural module of TRATTO includes two operational modes: the *Oracle evaluator*, which determines whether an oracle should be generated for a given unit under test, and the *Token selector*, which incrementally guides the construction of the oracle by selecting the most appropriate token at each generation step. Both components are implemented as tasks within a multitask deep learning model, enabling shared representations and efficient learning across related targets.

3.4.1 Oracle Evaluator

Although conceptually identified as a binary classification task (deciding whether or not to generate an oracle), the oracle evaluation is formulated as a masked token generation problem to leverage the strengths of auto-regressive models specialized for code generation and completion. This design choice enables the reuse of the same architectural backbone for both the *Oracle evaluator* and the *Token selector*, facilitating transfer learning and reducing training complexity.

The input to the *Oracle evaluator* follows the structured template reported in Listing 3.2. It consists of:

- The oracle type and corresponding Javadoc tag, when available (line 1).
- Two candidate tokens representing possible next steps: '`assertTrue(`' (indicating that oracle generation should proceed) or '`// No assertion possible`' (indicating that no oracle can be inferred.) (line 2).
- The mask token to fill out (line 4).
- The method under test, including its Javadoc documentation and source code (lines 6–8).

The model predicts which of the two candidate tokens should replace the mask, effectively making a binary decision about oracle generation.

```
1 // <oracle_type>: "<javadoc_tag>"
2 // Next possible tokens: ['assertTrue(', '// No assertion possible']
3 // Assertion:
4 <FILL_ME>
5
6 // Method under test:
7 <method_javadoc>
8 <method_source>
```

Listing 3.2. Input template for the Oracle Evaluator.

3.4.2 Token Selector

The *Token selector* extends the oracle evaluation paradigm to the iterative construction of test oracles. Like the *Oracle evaluator*, it is framed as a masked token generation task, but operates within a richer and more context-sensitive input structure (Listing 3.3).

The input template for the *Token selector* augments that of the *Oracle evaluator* with:

- A comprehensive list of candidate tokens representing all syntactically and semantically valid tokens that could extend the current partial oracle (line 2).
- The current partial oracle, which reflects the tokens generated in previous iterations (line 4).
- The method under test, including its Javadoc documentation and source code (lines 6–8).
- Additional contextual information, including method signatures and field declarations related to the candidate tokens, to support more informed decision-making by the model (lines 10–11).

Provided this enriched input, the model selects the token that produces a valid and semantically meaningful progression toward a complete oracle, when appended to the partial oracle.

Listing 3.4 provides a concrete example of the input presented to the *Token selector* during the construction of the oracle. When executed with Listing 3.4, the model generates an exceptional postcondition (line 1) where the partial oracle generated from the previous iterations is "`array.getClass().`" (line 4). Since the '`getClass()`' method returns an object of type '`Class`', the set of candidate tokens consists of non-private fields and methods of the '`Class`' class (line 2). Contextual information, such as method signatures (e.g., "`public native boolean isArray()` on line 25), helps the model distinguish between candidates. The model correctly selects the token "`isArray`' to replace the mask token, thereby extending the partial oracle and continuing the generation process.

```
1  // <oracle_type>: "<javadoc_tag>"
2  // Next possible tokens: [<next_possible_tokens>]
3  // Assertion:
4  assertTrue(<partial_oracle><FILL_ME>
5
6  // Method under test:
7  <method_javadoc>
8  <method_source>
9
10 // Additional context:
11 <method_signatures_and_field_declarations>
```

Listing 3.3. Input template for the Token Selector.

3.4.3 Multitask Learning Framework

By unifying the *Oracle evaluator* and *Token selector* within a single multitask model, TRATTO benefits from shared intermediate representations and enhanced generalization. This architecture allows the model to jointly learn when to generate oracles and how to construct them token-by-token, leveraging commonalities between the two tasks while maintaining task-specific output heads. The multitask formulation also enables more efficient use of training data, as both tasks contribute to refining the model's understanding of code semantics, documentation, and oracle structure.

```
1  // Exceptional postcondition: "@throws IllegalArgumentException if <code>array</code>
       isnot an array."
2  // Next possible tokens: ['equals', 'toString',    'isArray', 'getClassData',
       'getClassLoader', ...]
3  // Assertion:
4  assertTrue(array.getClass().<FILL_ME>;
5
6
7  // Method under test:
8  /**
9    * Constructs an ArrayListIterator that will
10   * iterate over the values in the specified array.
11   *
12   * @param array the array to iterate over
13   * @throws IllegalArgumentException if
14   * <code>array</code> is not an array
```

```
15    * @throws NullPointerException if
16    * <code>array</code> is <code>null</code>
17    */
18  public ArrayListIterator(final Object array){
19      super(array);
20  }
21
22  // Additional context:
23  public boolean equals(Object arg0)
24  public String toString()
25  public native boolean isArray()
26  Object getClassData()
27  public ClassLoader getClassLoader()
28  ...
```

Listing 3.4. Input to the model acting as Token Selector.

3.5 Data Construction and Training

The neural model underlying both the *Oracle evaluator* and *Token selector* requires training data structured in two complementary forms: (i) a dataset of complete axiomatic oracles and (ii) a dataset of individual tokens that we obtain by decomposing those oracles into their corresponding lexical tokens. Figure 3.2 illustrates the pipeline to construct the two datasets. We started with an existing corpus of procedure specifications, which we subsequently improved and enriched through manual inspection and automated augmentation techniques. The *Token collector* and *Token filter* of Tratto's symbolic module disaggregated oracles into token-level samples, producing the tokens dataset. Both datasets are publicly available in the replication package of the work [3].

3.5.1 Procedure Specifications Dataset

The foundation of TRATTO's training data is the procedure specifications dataset provided by Blasi et al. [8]. This corpus comprises 3,150 Javadoc preconditions (extracted from "`@param`' tags), regular postconditions (extracted from "`@param`' tags), and exceptional postconditions that characterize the intended semantics of program units, together with the corresponding target oracles. These specifications are encoded in tuples of the form $\langle (u, jt), o \rangle$, where u represents a *unit* under test, jt denotes a *Javadoc tag* extracted from the unit's documentation, and o is an executable Boolean expression formally describing that tag, and suitable for use as a test oracle. Additionally, the original dataset includes 23,397 Javadoc tags for which the symbolic approach of Blasi et al. was unable to generate corresponding axiomatic oracles.

3.5.2 Oracles Dataset

We created an initial oracles dataset by combining 3,150 procedure specifications from Blasi et al. labeled as *positive* samples (tuples of tags associated to oracles successfully generated) with 3,150 randomly procedure specifications labeled as *negative* samples (comments lacking oracles). Since the authors reported an average 92% of precision and 83% of recall, we conducted a comprehensive manual inspection across all 6,300 instances to correct errors, such as incorrectly generated or missed oracles (step ① in Figure 3.2). Additionally, we integrated 222

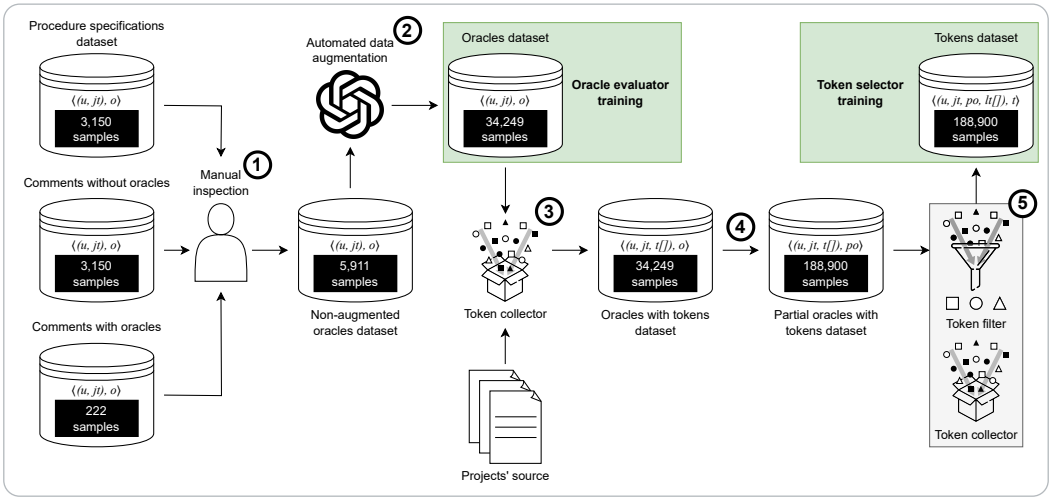

Figure 3.2. Data collection process to fine-tune the neural component of Tratto as Oracle evaluator and Token selector

manually verified oracles from various open-source Java projects on GitHub to further enhance dataset quality and diversity.

This process yielded an initial dataset of 5,911 samples, 4,582 of which were positive (comments with related oracles) and 1,329 were negative (comments without oracles). The number of positive cases increased because, upon careful analysis, we identified many of the initially non-matching comments with missing valid oracles that we manually added.

To further expand the dataset and improve the model's generalization capabilities, we generated semantically equivalent versions of the Javadoc comments in the tuples, using ChatGPT. We prompted the model to produce alternative phrasings (up to five per original comment) that preserved the intended specification semantics while varying the linguistic expression (step ②). Listing 3.5 provides an illustrative set of equivalent Javadoc tags produced through this augmentation strategy. The final augmented oracles dataset comprises 34,249 samples: 23,392 positive instances (with oracles) and 10,857 negative instances (without oracles).

```
@return the sum {@code a + b}. // Original
@return the total value of {@code a + b}
@return the result of adding {@code a} and {@code b}
@return the outcome of summing {@code a} and {@code b}.
@return the value obtained by adding {@code a} and {@code b}.
@return the sum of {@code a} and {@code b}
```

Listing 3.5. Equivalent Javadoc tags generated with ChatGPT.

3.5.3 Tokens Dataset

Building upon the oracles dataset, we constructed a tokens dataset to support the training of the *Token selector* in the token prediction task. We generated the dataset through a systematic disaggregation process (steps ③-⑤ in Figure 2), relying on the *Token collector* to analyze the

project's source code and the unit under test related to each tuple in the oracles dataset to extract generic tokens potentially relevant for the generation of the target oracle (step ③, such as class names `HashBiMap`), constants (e.g., `CollectionUtils.EMPTY_COLLECTION`), and method names (e.g., `.toString()`). This procedure yields tuples in the form ⟨*(u, jt, t[]), o*⟩, where *t[]* refers to the *list of all possible tokens* that could be initially used to start building the oracle *o*.

Next, the pipeline systematically decomposed each oracle into a series of partial oracles *po* (step ④), representing the stepwise states encountered during incremental token generation. For each partial oracle *po*, the *Token filter* pruned the list of tokens previously collected by the *Token collector*, enforcing syntactic and semantic validity according to the grammar and type system, ultimately producing a list *lt[]* of legal next tokens (step ⑤). This workflow led to the final dataset of tokens in the form of ⟨*(u, jt, po, lt[]), t*⟩ tuples, where *po* denotes a *partial oracle*, *lt[]* refers to the *list of legal tokens* that could possibly follow the partial oracle (e.g., '`0`', '`1`', and '`SomeClass`' could follow "`result >`"), and *t* is the target next token following the partial oracle (*t* must be one of the tokens from the aforementioned list of legal tokens *lt[]*). The tokens dataset contained 188,900 samples in total.

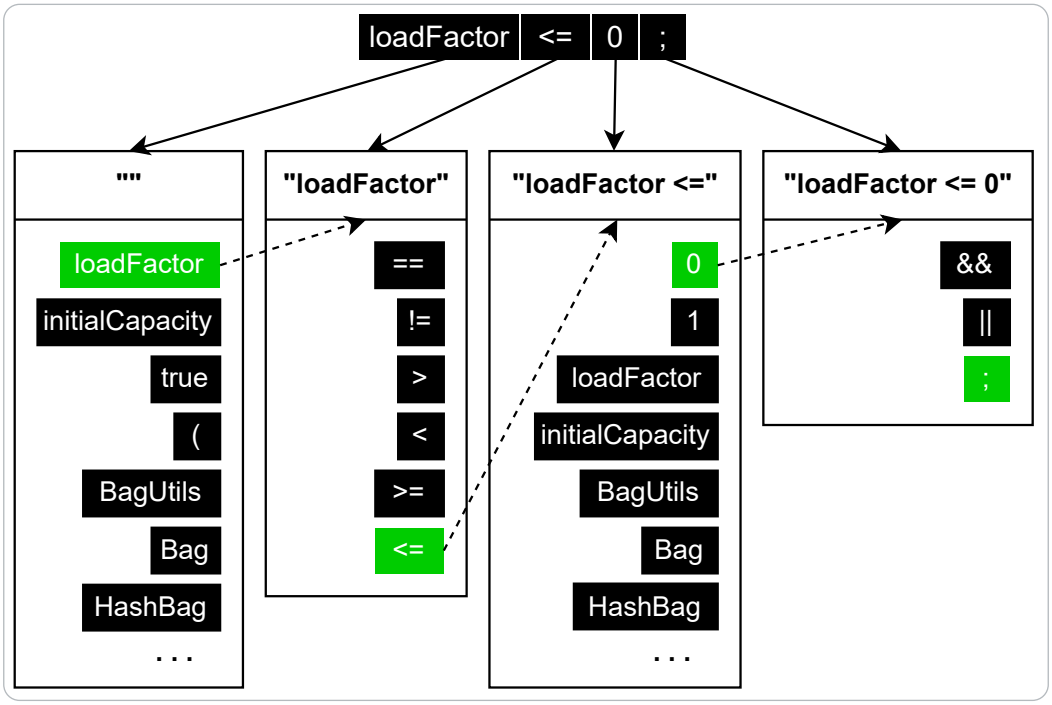

Figure 3.3. Converting one oracle sample into four token samples. Oracle (o) at the top; partial oracles (po) and legal tokens ($lt[]$) on top and bottom white boxes, respectively; next tokens (t) in green.

Figure 3.3 illustrates the process of decomposing oracles into tokens samples. Let us consider the oracle "`loadFactor <= 0;`", which encodes an exceptional postcondition, according to which, if the argument '`loadFactor`' is less than or equal to zero, the method should throw an exception. Since this oracle contains four tokens, the process produces four partial oracles

through consecutive iterations: In the initial state (with an empty partial oracle), the candidate tokens include method parameters (e.g., '`loadFactor`, '`initialCapacity`'), parentheses, and class names for static calls. Once '`loadFactor`" appears as the partial oracle, only comparison operators (like '<=', '>', '==) are valid, as the *Token filter* excludes other operators that are either forbidden by the grammar or semantically incompatible with numeric types. The process iteratively narrows the candidate token set based on the evolving partial oracle, until reconstructing the entire oracle.

This token-level perspective empowers the neural model to capture fine-grained generation dynamics, learning both from the formal symbolic constraints of the grammar and from distributional patterns present in the training data.

3.6 Evaluation

Our experimental evaluation addresses the following research questions:

RQ-1: *How do different LLM code models compare when used to evaluate oracles and select tokens?* This question evaluates different code models to identify the most suitable backbone for the neural module of TRATTO.

RQ-2: *What is the contribution of the symbolic module and the multitask model to the overall performance of* TRATTO*?* This question employs ablation studies to assess TRATTO's performance when operating (i) without the symbolic module (as a purely neural model lacking the token-by-token generation approach), and (ii) without the multitask architecture (using separate models for the *Oracle evaluator* and *Token selector*).

RQ-3: *What is the effectiveness of* TRATTO *in generating axiomatic test oracles and how does it compare with state-of-the-art neural and symbolic techniques?* This question benchmarks TRATTO against representative state-of-the-art symbolic and neural baseline, using a curated ground-truth dataset of axiomatic test oracles.

RQ-4: *How robust is* TRATTO *to documentation variations?* Building upon the results of RQ-3, this question considers the set of oracles that all approaches correctly predict, applies systematic variations to their associated documentation, and measures each approach's robustness in terms of oracle generation accuracy under modified documentation.

RQ-5: *How effective are the generated oracles for improving test suites?* This question integrates the oracles generated by TRATTO into test suites automatically produced with EvoSuite and measures their effectiveness through improvements in mutation score.

3.6.1 RQ-1: Code Models Comparison

RQ-1 investigated the suitability of large language models for the two core tasks performed by the neural module implemented in TRATTO: *evaluating oracles* and *predicting the next oracle token*. We experimentally compared three code models for coding task available at the time—Code Gemma by Google [62], StarCoder2 by BigCode [45], and Code Llama by Meta [59]—to determine the most effective LLM for these tasks. The experiment formulates both tasks as code infilling problems (see Listings 3.2 and 3.3) and selects these models based on the infilling support, training on large-scale code corpora, and popularity among researchers and practitioners for code-related tasks. All experiments employ the 7B-parameter version of each model.

The training and validation dataset comprises 223,149 samples (34,249 oracles + 188,900 tokens). The evaluation allocated all samples belonging to a single Java project (Guava, 30,891 samples, 14% of the total) for validation and uses the remaining 192,258 samples (86%) for training, to prevent data leakage (since some oracle and token samples from the same project differ only in rephrased Javadoc tags due to the augmentation process described in Section 3.5)

The experiment trained all models for two epochs, as empirical analysis revealed that accuracy plateaus beyond this point. The input length spanned 2,048 transformer tokens, while the output length extended to 32 transformer tokens (note that a single lexical token, such as a variable name like '`millis2secs`', typically decomposes into multiple transformer tokens, e.g., '`mill`'-'`is`'-'2'-'`se`'-'`cs`' comprises five tokens). Training employed default hyperparameters for all models.

Code Gemma, StarCoder2, and Code Llama achieved accuracy levels of 58%, 69%, and 91%, respectively. Code Llama substantially outperformed both Code Gemma and StarCoder2. This performance gap likely stems from differences in pre-training data and training specialization. Code Llama was pre-trained primarily on natural language and actual code, with additional specialization for long-context inputs, which aligns well with the tasks implemented in TRATTO. In contrast, StarCoder2 learns from heterogeneous data sources such as GitHub issues, pull requests, and Jupyter notebooks, whereas Code Gemma focuses on English-language data from open-source math datasets and synthetically generated code. Both Code Gemma and StarCoder2 rely on general data and synthetic code that is less relevant for generating oracles that makes heavy reference to natural language documentation.

When evaluating complete oracle generation (where all tokens in an oracle sequence must be correctly predicted) TRATTO instantiated with Code Llama correctly generates 69% of axiomatic oracles on the Guava validation set, producing 3,337 correct oracles out of 4,865 cases (this dataset includes non-oracle samples, which count as correctly predicted when the *Oracle evaluator* properly judges that no oracle should be generated).

Answer to RQ-1: Code Llama achieves 91% accuracy for evaluating oracles and selecting tokens, substantially outperforming both Code Gemma (58%) and StarCoder2 (69%).

3.6.2 RQ-2: Ablation Studies

RQ-2 examines the contributions of the symbolic module and the multitask model implemented in TRATTO to the overall oracle generation performance through two ablation studies that systematically remove these components.

The first study removes the token-by-token oracle generation approach enabled by the symbolic module. In this configuration, the neural module either infers the complete oracle or indicates the impossibility to generate an oracle from the given javadoc comment, relying solely on the unit under test. The study reserves Guava oracles (4,865 samples) for validation and fine-tunes the Code Llama 7B model on the remaining 29,384 samples of the oracles dataset using default hyperparameters for two epochs, in line with the training setup. The purely neural model generates 3,049 correct oracles out of 4,865, achieving 63%–6 percentage points lower than Tratto's 69% accuracy. The McNemar test confirms the statistical significance of the difference (p-value < 0.001, Odds Ratio $= 2.57$), demonstrating that the symbolic module and token-by-token approach substantially improve axiomatic oracle generation performance.

The second study removes the multitask architecture and fine-tunes two separate models: a *Oracle evaluator* and a *Token selector*, both based on Code Llama 7B with default hyperpa-

rameters for two epochs. We trained the *Oracle evaluator* on 29,384 oracle samples (86% of all oracles, excluding Guava), and the Selector on the corresponding 162,874 token samples (86% of all tokens). The evaluation measures the combined capability of both models to correctly predict the 4,865 Guava validation oracles, requiring the *Oracle evaluator* to correctly determine whether the oracle generation should proceed and, if so, the *Token selector* to correctly predict all oracle tokens. This approach achieves 66% accuracy (3,213 out of 4,865 oracles), 3 percentage points below the performance registered by TRATTO. The difference is statistically significant (p-value < 0.001, Odds Ratio $= 1.55$), highlighting the advantages of the multitasking learning framework for the joint evaluation of oracles and token prediction.

The analysis of the oracles that TRATTO generates successfully while the ablated approaches fail reveals that the ablated models either produce false positives, generating oracles where none should exist, or generate incorrect oracles. As an example of false positives, the purely neural model incorrectly generates the precondition '`array != null`' from the Javadoc tag "*@param array an array of @code short values, possibly empty*". The *Oracle evaluator* of the non-multitask model generates a precondition from the tag "*@param defaultValue the value provided for inputs absent in map keys*", which expresses no precondition.

Answer to RQ-2: The purely neural and non-multitask approaches achieve 63% and 66% accuracy, respectively, both falling below Tratto's 69% accuracy. These results confirm that both the symbolic module and the multitask model contribute meaningfully to TRATTO's performance.

3.6.3 RQ-3: Oracle Generation

RQ-3 compares the effectiveness of TRATTO against state-of-the-art neural and symbolic approaches for generating axiomatic oracles. This section details the ground-truth dataset, the comparative approaches employed, the evaluation metrics, the experimental setup, and the results obtained.

Ground Truth Dataset

To ensure a fair comparison with state-of-the-art approaches, the evaluation uses a manually curated ground-truth dataset of axiomatic test oracles. The dataset includes oracles from Defects4J, excluding two projects—*Apache Commons Math* and *Apache Commons Collections*—that are part of TRATTO's training set. For each of the remaining 15 projects, the study systematically selects 10 Java classes by sorting them according to character count and choosing those evenly spaced within the 5% to 95% range (i.e., classes at the 5th, 15th, 25th percentiles, etc.). We manually extract all possible axiomatic oracles from every method of each selected class.

The dataset contains 389 axiomatic oracles (positive samples) spanning 274 methods across 150 classes from 15 projects. For each method, the study also generates a negative sample if the method admits no precondition, postcondition, or exceptional postcondition. For instance, a method encoding two preconditions yields two positive samples (the two preconditions) plus two negative samples (a non-postcondition and a non-exceptional-postcondition). The dataset collects in total 496 negative samples. At least two authors reviewed each sample.

Comparison Approaches

The evaluation compares TRATTO against Jdoctor [8] and GPT-4 [53] as representative symbolic

and neural approaches, respectively. Jdoctor generates axiomatic oracles (preconditions, normal and exceptional postconditions) from the Javadoc tags '`@param`', '`@return`', and '`@throws`' by applying pattern, lexical, and semantic matching techniques. Jdoctor outperforms other approaches such as Toradocu and @tComment.

State-of-the-art neural approaches for oracle generation [67, 72, 19] produce concrete test oracles (assertions on specific inputs) rather than axiomatic oracles (predicates on variables valid for all inputs), making the direct comparison with TRATTO impossible. The study compares TRATTO against GPT-4 (model GPT-4o) as a representative neural approach. We enhance GPT-4 with few-shot learning combined with Chain-of-Thought prompting to fully leverage its understanding capabilities, providing three examples demonstrating how and when to generate axiomatic oracles from Java methods using a step-by-step approach. We ask GPT-4 to generate oracles for the ground-truth dataset, method by method.

Metrics

We comparatively evaluate TRATTO by computing *true positives* (correctly predicted oracles), *true negatives* (instances for which no oracle should be generated, and none is generated), *false positives* (instances for which no oracle should be generated, but one is generated, or wrongly generated oracles), and *false negatives* (oracles not generated, while one should be generated).

Our study classifies incomplete oracles capturing only a subset of correct behavior as false positives, since they partially miss the reference oracle's semantics and fail for some test cases. For example, '`result != null`' represents an incomplete oracle for a method returning a positive '`Integer`', correctly capturing only partial expected behavior, while '`result >= 0`' constitutes a wrong oracle for the same method, since zero is not positive.

The evaluation also includes the computation of accuracy (percentage of correct predictions out of the total number of predictions), precision (percentage of correctly generated oracles out of the total number of generated oracles), recall (percentage of correctly generated oracles out of the total number of actual oracles in the dataset), and F1-score (weighted harmonic mean of precision and recall). The accuracy measures the overall performance without differentiating between false positives and negatives. The precision measures the ability to avoid false positives. The recall reflects the ability to capture all relevant instances, avoiding false negatives. The F1-score combines precision and recall into a single measure balancing false positives and negatives, proving especially useful for unbalanced datasets where the positive class is rare.

Training and Evaluation Setup

We fine-tuned TRATTO using the complete dataset (223k samples) with the same setup applied in the previous experiments. We downloaded and set up the most recent version of Jdoctor from the publicly available GitHub repository (commit d76899f). Jdoctor does not need training, as it is based on a set of heuristic rules and patterns. We generated as many axiomatic oracles as possible for all methods in the ground-truth dataset, for both TRATTO and Jdoctor. We performed similarly for GPT4, generating as many axiomatic oracles as possible for all methods, by crafting a prompt per method, making an API call per prompt, and collecting all responses, which we manually analyzed.

Results

Table 3.1 reports the *accuracy* (row A), *precision* (P), *recall* (R), and *F1-score* (F1) computed

Table 3.1. Accuracy (A), precision (P), recall (R), and F1-score ($F1$) for all approaches on ground-truth dataset.

		closure-compiler	commons-cli	commons-codec	commons-compress	commons-csv	commons-jxpath	commons-lang	gson	jackson-core	jackson-databind	jackson-dataformat	jfree-chart	joda-time	jsoup	mockito	Total
TRATTO	A	56%	83%	61%	74%	91%	62%	75%	80%	68%	58%	67%	77%	81%	60%	67%	**73%**
	P	N/A	60%	29%	82%	88%	N/A	61%	70%	43%	33%	N/A	90%	83%	50%	N/A	**72%**
	R	0%	75%	24%	53%	97%	0%	67%	70%	33%	3%	0%	60%	78%	5%	0%	52%
	F1	N/A	67%	26%	64%	92%	N/A	64%	70%	37%	6%	N/A	72%	80%	8%	N/A	**61%**
Jdoctor	A	56%	72%	56%	73%	79%	62%	60%	64%	67%	61%	67%	47%	59%	57%	67%	61%
	P	N/A	100%	33%	100%	96%	N/A	0%	6%	33%	100%	N/A	N/A	74%	0%	N/A	62%
	R	0%	16%	4%	41%	68%	0%	0%	7%	11%	3%	0%	0%	33%	0%	0%	**16%**
	F1	N/A	29%	7%	58%	79%	N/A	N/A	6%	17%	6%	N/A	N/A	46%	N/A	N/A	**25%**
GPT4	A	18%	26%	37%	42%	55%	5%	53%	19%	47%	35%	50%	35%	53%	41%	50%	**40%**
	P	12%	7%	20%	27%	40%	2%	31%	3%	22%	13%	0%	27%	42%	23%	33%	**24%**
	R	100%	100%	92%	100%	92%	100%	88%	60%	100%	100%	N/A	92%	96%	67%	100%	**89%**
	F1	22%	12%	32%	42%	56%	4%	46%	6%	36%	24%	N/A	42%	58%	34%	50%	37%

Table 3.2. True/false positives/negatives ($TP/TN/FP/FN$) for all approaches on ground-truth dataset.

Project	M	O	NO	TRATTO								Jdoctor								GPT4							
				TP		TN		FP		FN		TP		TN		FP		FN		TP		TN		FP		FN	
closure-compiler	3	4	5	0	(0%)	5	(56%)	0	(0%)	4	(44%)	0	(0%)	5	(56%)	0	(0%)	4	(44%)	2	(12%)	1	(6%)	14	(0%)	0	(82%)
commons-cli	6	6	12	3	(16%)	12	(67%)	2	(6%)	1	(11%)	1	(5%)	12	(67%)	0	(0%)	5	(28%)	1	(5%)	4	(21%)	14	(74%)	0	(0%)
commons-codec	19	24	33	4	(7%)	32	(54%)	10	(22%)	13	(17%)	1	(2%)	31	(54%)	2	(4%)	23	(40%)	11	(15%)	16	(22%)	45	(62%)	1	(1%)
commons-compress	12	17	20	9	(24%)	19	(50%)	2	(5%)	8	(21%)	7	(19%)	20	(54%)	0	(0%)	10	(27%)	12	(21%)	12	(21%)	33	(58%)	0	(0%)
commons-csv	16	35	23	30	(52%)	23	(40%)	4	(7%)	1	(2%)	23	(39%)	23	(39%)	1	(2%)	11	(18%)	24	(28%)	23	(27%)	36	(42%)	2	(2%)
commons-jxpath	4	5	8	0	(0%)	8	(62%)	0	(0%)	5	(38%)	0	(0%)	8	(62%)	0	(0%)	5	(38%)	1	(2%)	2	(4%)	49	(94%)	0	(0%)
commons-lang	53	64	103	37	(22%)	89	(53%)	24	(14%)	18	(11%)	0	(0%)	101	(61%)	2	(1%)	64	(38%)	38	(20%)	62	(33%)	83	(44%)	5	(3%)
gson	27	30	51	19	(23%)	46	(57%)	8	(10%)	8	(10%)	1	(1%)	51	(63%)	16	(20%)	13	(16%)	3	(3%)	18	(16%)	88	(79%)	2	(2%)
jackson-core	10	10	20	3	(10%)	18	(58%)	4	(13%)	6	(19%)	1	(3%)	19	(63%)	2	(7%)	8	(27%)	5	(15%)	11	(32%)	18	(53%)	0	(0%)
jackson-databind	24	30	48	1	(1%)	42	(57%)	2	(3%)	29	(39%)	1	(1%)	46	(60%)	0	(0%)	29	(39%)	13	(10%)	33	(25%)	84	(65%)	0	(0%)
jackson-dataformat	1	1	2	0	(0%)	2	(67%)	0	(0%)	1	(33%)	0	(0%)	2	(67%)	0	(0%)	1	(33%)	0	(0%)	2	(50%)	2	(50%)	0	(0%)
jfree-chart	25	46	40	26	(0%)	40	(47%)	3	(3%)	17	(20%)	0	(0%)	40	(47%)	0	(0%)	46	(53%)	34	(24%)	17	(12%)	90	(62%)	3	(2%)
joda-time	40	71	63	52	(39%)	56	(42%)	11	(8%)	15	(11%)	23	(17%)	56	(42%)	8	(6%)	47	(35%)	50	(33%)	31	(20%)	70	(46%)	2	(1%)
jsoup	33	45	66	2	(2%)	65	(59%)	2	(2%)	42	(38%)	0	(0%)	64	(57%)	4	(3%)	45	(40%)	22	(16%)	36	(25%)	73	(51%)	11	(8%)
mockito	1	1	2	0	(0%)	2	(67%)	0	(0%)	1	(33%)	0	(0%)	2	(67%)	0	(0%)	1	(33%)	1	(25%)	1	(25%)	2	(50%)	0	(0%)
Total	274	389	496	186	**(21%)**	459	(52%)	72	(8%)	169	(19%)	58	**(7%)**	480	**(54%)**	35	**(4%)**	312	**(35%)**	217	(18%)	269	**(22%)**	701	**(58%)**	26	**(2%)**

for each approach (TRATTO, Jdoctor, and GPT-4) across each project (columns *closure-compiler* through *mockito*) and overall (column *Total*). The table highlights the best *Total* values in green and the worst in red.

TRATTO outperforms both Jdoctor and GPT-4 in terms of accuracy (73% for TRATTO, 61% for Jdoctor, 40% for GPT-4), precision (72% for TRATTO, 62% for Jdoctor, 24% for GPT-4), and F1-score (61% for TRATTO, 25% for Jdoctor, 3% for GPT-4). TRATTO's recall (52%) exceeds Jdoctor's (16%) but falls below GPT-4's (89%). These results indicate that TRATTO infers more correct predictions than both Jdoctor and GPT-4 (accuracy) with minimal impact from wrong results (precision). GPT-4's superior recall demonstrates its capability to generate a higher proportion of oracles from the ground truth than TRATTO (an expected outcome given GPT-4's general-purpose nature and superior generalization capabilities as a large-scale LLM). However, this higher recall comes at the cost of substantially more false positives (lower precision). TRATTO's fine-tuning enables more precise evaluation of whether oracle generation should proceed and what form the oracle should take. The F1-score effectively summarizes the improvement of the token-by-token neuro-symbolic approach implemented in TRATTO over Jdoctor's symbolic approach and GPT-4's neural approach. Approach performance varies greatly across projects, confirming dependency on comment and code quality.

Table 3.2 provides detailed breakdowns for each project and approach, reporting the number of methods (column M), ground-truth oracles (O), non-oracle ($negative$) instances (NO), and true/false predictions (TP, TN, FP, FN) for all approaches and projects, plus totals. For each metric, the table reports both total prediction counts and percentages over total instances ($O+NO = 389+496 = 885$). Note that $TP+TN+FP+FN$ may exceed 885 since approaches

can generate arbitrary numbers of false positives.

The *Total* row of Table 3.2 highlights the best performance in green and worst in red. TRATTO generates the highest rate of correct oracles (total true positive rate 21%) while avoiding oracle generation where inappropriate in many cases (total true negative rate 52%, just below Jdoctor's best result of 54%). TRATTO also performs well in terms of false alarms and missed oracles, with low false positive/negative rates of 8% and 19%, respectively. Jdoctor performs slightly better than TRATTO for true negatives (54% vs. 52%) and false positives (4% vs. 8%) but performs worst among the three approaches for true positives (7%) and false negatives (35%). GPT-4 presents an excellent false negative rate (only 2%) but poor true positive (18%), true negative (22%), and false positive rates (58%). These true/false positive/negative rates confirm the best performance of TRATTO among the three approaches, well-summarized by the superior *F1-score* in Table 3.1.

Overall, TRATTO generates 186 out of 389 ground-truth oracles and demonstrates strong capability to discern when oracle generation should proceed, with relatively few wrong oracles (72 out of 885 predictions). Thus, TRATTO can significantly reduce manual oracle generation effort without imposing substantial overhead for identifying and discarding incorrect oracles.

TRATTO generates three times more correct oracles than Jdoctor (186 vs. 58). This result confirms the impact of developers' jargon: Jdoctor exploits classic natural language processing and semantic matching, performing well with precise natural language comments, but struggling with imprecise comments common in Javadoc documentation. The neuro-symbolic approach implemented in TRATTO demonstrates much greater tolerance for comment imprecision, handling many more cases than Jdoctor. Results depend heavily on comment and code quality. Jdoctor generates a fair number of oracles for *commons-csv* (23 oracles while TRATTO generates 30) but produces no oracles for *jfree-chart*, where TRATTO generates 26 correct oracles.

GPT-4 infers 217 correct oracles, a modest increment over TRATTO (186 correct oracles). This reflects the substantial difference in neural component scale (GPT-4 features hundreds of billions of parameters versus Code Llama's 7 billion parameters underlying TRATTO). However, GPT-4 generates 701 false positives—nearly 10 times more than TRATTO (72), accounting for more than half of all predictions (58%). This significant false positive rate showcases the main difference between GPT-4's pure neural approach and TRATTO's neuro-symbolic approach: the symbolic component of TRATTO discriminates valid from invalid results, while its neural module is explicitly optimized to determine when oracle generation is appropriate (*Oracle evaluator*). Thus, TRATTO generates fewer valid results than GPT-4 but also a much more limited number of invalid results, while GPT-4 neither checks validity nor ensures compilability, often generating oracles where none should exist. GPT-4's large false positive volume greatly reduces practical applicability, requiring massive human effort to prune results (as demonstrated in widely cited studies [13, 39, 60] reported in [32]).

GPT-4 produces few false negatives (26, 2% of total predictions), while TRATTO and Jdoctor fail to generate oracles for 169 and 312 ground-truth oracles (19% and 35% of total predictions), respectively. GPT-4's strong false negative performance balances poor false positive results: GPT-4 almost always attempts oracle generation, consequently generating many wrong oracles (false positives) while missing very few cases (false negatives). Conversely, both TRATTO and Jdoctor identify cases where no oracle should generate, producing fewer wrong oracles. The overall performance combines false predictions: GPT-4 generates 727 total false results (701 FP + 26 FN), Jdoctor 347 (35 + 312), and TRATTO only 241 (72 + 169).

The second '`@throws` tag in Listing 6 (line 6) exemplifies Tratto's capability to infer correct exceptional postcondition oracles from Javadoc tags that both Jdoctor and GPT-4 fail to

interpret. This tag contains an implicit reference (*"this method is called on a closed result set"*) to a method (`isClosed`) of a class (`ResultSet`) from an external library (`java.sql`): Tratto's symbolic module retrieves contextual information about the external class from the method signature (line 9, `resultSet` method parameter) and feeds the neural model additional information to produce the correct axiomatic oracle '`resultSet.isClosed();`'. Jdoctor cannot infer an oracle from this Javadoc tag since it falls outside the set of rules and patterns, resulting in a false negative. GPT-4 generates the precondition '`resultSet != null;`', claiming the parameter must not be null, despite neither documentation nor contextual information indicating this requirement. GPT-4 also generates three exceptional postconditions ('`resultSet == null ? NullPointerException;`', '`IOException may be thrown;`', and '`SQLException may be thrown if resultSet is closed or if there is a database access error`') – either wrong or non-compilable. This example highlights how pure neural models struggle to infer non-trivial oracles, may generate non-compilable oracles due to lacking precise grammar, and produce non-negligible false positives.

```
1  /**
2  * Prints headers for a result set based on its metadata.
3  *
4  * @param resultSet The ResultSet to query for metadata.
5  * @throws IOException If an I/O error occurs.
6  * @throws SQLException If a database access error occurs or this method is called on a
       closed result set.
7  * @since 1.9.0
8  */
9  public synchronized void printHeaders(final ResultSet resultSet) throws IOException,
       SQLException {
10   printRecord((Object[]) format.builder().setHeader(resultSet).build().getHeader());
11 }
```

Listing 3.6. Documentation and implementation of method printHeaders from CSVPrinter.

Manual inspection of the oracles generated by TRATTO and Jdoctor reveals only one oracle that Jdoctor generates while TRATTO does not–an exceptional postcondition stating a method parameter must be of a certain type (via the '`instanceof`' operator). The hypothesis suggests TRATTO's training set lacked sufficient oracles of this kind, leading the *Oracle evaluator* to infer the lack of necessity of generating oracles. Conversely, TRATTO generates significantly more oracles than Jdoctor, as Jdoctor's symbolic matching often overlooks similar cases. For example, TRATTO successfully generates exceptional oracles '`source == null;`' and '`bigInteger == null;`' from the Javadoc comments *"the parameter passed to this method is null"* in Listing 3.7 (lines 14-15) and *"@throws NullPointerException if null is passed in"* in Listing 3.8 (line 8), respectively.

Answer to RQ-3: TRATTO achieves a 61% *F1-score* in generating axiomatic oracles, significantly exceeding Jdoctor (25%) and GPT-4 (37%). TRATTO generates a high number of oracles (186, 3× more than Jdoctor) while incurring few false positives (72, 10× less than GPT-4).

3.6.4 RQ-4: Robustness to Documentation Variations

RQ-4 evaluates the robustness of TRATTO in generating axiomatic oracles under varying documentation quality, and compares the robustness of TRATTO with both Jdoctor and GPT-4. The study selects all oracles that all three approaches successfully inferred in **RQ-3**(3.6.3), systematically produces variations of their associated documentation, and computes the proportion of oracles that each approach can still correctly generate with modified documentation.

The study identifies 55 oracles that all three approaches successfully inferred from eight ground-truth projects. For each oracle, the evaluation identifies the Javadoc tag from which the oracle derives and generates four variations using GPT-4 with systematic criteria: (i) replacing words with synonyms or rephrasing while preserving meaning; (ii) changing sentence order; (iii) introducing grammatical mistakes or typos; and (iv) reducing explicitness. Listing 3.9 shows examples of generated variations.

```
/**
* Decodes an "encoded" Object and returns a
* "decoded" Object. Note that the implementation of
* this interface will try to cast the Object
* parameter to the specific type expected by a
* particular Decoder implementation. If {@link
* ClassCastException} occurs this decode method will
* throw a DecoderException.
*
* @param source the object to decode
* @return a 'decoded" object
* @throws DecoderException a decoder exception can
* be thrown for any number of reasons. Some good
* candidates are that the parameter passed to this
* method is null, a param cannot be cast to the
* appropriate type for a specific encoder.
*/
Object decode(Object source) throws DecoderException;
```

Listing 3.7. Documentation and implementation of method decode from Decoder.

```
/**
* Encodes to a byte64-encoded integer according to
* crypto standards such as W3C's XML-Signature.
*
* @param bigInteger a BigInteger
* @return A byte array containing base64 character
* data
* @throws NullPointerException if null is passed in
* @since 1.4
*/
public static byte[] encodeInteger(final BigInteger
      bigInteger) {
  Objects.requireNonNull(bigInteger, "bigInteger");
  return encodeBase64(toIntegerBytes(bigInteger),
          false);
}
```

Listing 3.8. Documentation and implementation of method encodeInteger from Base64.

Table 3.3. Robustness to documentation variations.

Approach	Synonyms	Order	Typos	Explicitness	Total
TRATTO	53 (96%)	54 (98%)	54 (98%)	34 (62%)	195 (89%)
Jdoctor	16 (29%)	29 (53%)	4 (7%)	0 (0%)	49 (22%)
GPT4	52 (95%)	54 (98%)	53 (96%)	49 (89%)	208 (95%)

We manually checked the correctness for all the oracles generated with each approach across the 220 variations (55 oracles × 4 variations). Table 3.3 reports the number and percentage of the correctly generated oracles per approach. TRATTO generates approximately 90% of correct oracles across documentation variations, and closely matches GPT-4 (95%) despite relying on a much smaller model. TRATTO's performance matches GPT-4's performance across all variation types but 'less explicit description'. The result is not surprising, since 'less explicit descriptions' do not explicitly provide oracle-relevant information, as for instance, line 6 in Listing 3.9. Jdoctor proves to be the least robust approach, achieving only 22% correct oracles across variations and producing no oracles for less explicit descriptions.

```
1 (object == null) == false; // Generated oracle
2 @param object  the object to convert, must not be null // Original Javadoc tag
3 @param object the item to transform, should not be null // Synonyms or rephrasing
4 @param object Must not be null, this is the object that needs conversion. // Changed order
5 @param object the obect to convirt, musn't be nul // Grammatical mistakes or typos
6 @param object an element to be used, should be valid // Less explicit
```

Listing 3.9. Rephrased Javadoc tags generated with ChatGPT.

Answer to RQ-4: TRATTO is highly robust to documentation variations. It correctly generates 195 out of 220 oracles (89%), with a performance comparable to GPT-4 (208, 95%) and substantially higher than Jdoctor (49, 22%).

Table 3.4. Mutation score of test suites.

Project	Implicit oracles			Implicit/regression oracles		
	EvoSuite	TRATTO	Jdoctor	EvoSuite	TRATTO	Jdoctor
commons-cli	21%	**28%**	22%	61%	61%	61%
commons-codec	42%	**59%**	46%	71%	**75%**	71%
commons-compress	20%	**22%**	20%	41%	41%	41%
commons-csv	5%	5%	5%	13%	13%	13%
gson	22%	**30%**	24%	67%	**68%**	67%
joda-time	38%	**47%**	39%	78%	**79%**	78%
Total	25%	**32%**	26%	55%	**56%**	55%

3.6.5 RQ-5: Application to Software Testing

RQ-5 addresses the impact of the oracles on the test suite, by comparing the mutation scores with and without oracles. The exploratory study develops a script to automatically insert generated oracles into test cases containing calls to methods for which oracles exist. Oracles causing test case failures are discarded, though further examination may reveal bug-revealing tests, strengthening oracle usefulness. The mutation score computation uses the PIT mutation testing tool [15]. The study measures oracle impact as the difference between test suite mutation scores with and without oracles.

The evaluation excludes GPT-4 from this experiment due to its large volume of non-compilable oracles (552 out of 1,213 in **RQ-3** 3.6.3), which prevents automating oracle insertion into test suites.

EvoSuite generates test suites for the 10 classes per project from the ground-truth dataset for which oracles were generated in **RQ-3**. The study could not run EvoSuite on five projects (*closure-compiler*, *jackson-core*, *jackson-databind*, *jackson-dataformat*, and *mockito*) due to incompatibility issues. Three projects for which Jdoctor generated no ground-truth oracles (*jfreechart*, *commons-lang*, and *jsoup*), and one project for which neither TRATTO nor Jdoctor generated oracles (*commons-jxpath*) were excluded, since without oracles, test suites and mutation scores remain unchanged. Focusing on projects where both TRATTO and Jdoctor generate oracles ensures fair comparison.

The evaluation assesses improvement with automatically generated oracles on test suites containing implicit oracles only and those containing both implicit and regression oracles. The latter represents default EvoSuite-generated test suites, while the former presents a more realistic scenario where regression oracles are unavailable or cannot be assumed correct. Creating these test suites simply wraps EvoSuite-generated assertions in try-catch blocks.

Table 3.4 shows the mutation scores for the test suites with EvoSuite, TRATTO, and Jdoctor, with and without regression oracles. The oracles generated by TRATTO improve mutation scores relative to EvoSuite's implicit oracles for 5 out of 6 projects, ranging from 2% (*commons-compress*) to 17% (*commons-codec*), averaging 7% increase across all projects. Oracles also improve mutation scores with EvoSuite's regression oracles for 3 out of 6 projects, averaging 1% increase. This lower increase is expected since EvoSuite's regression oracles make concrete behavioral assumptions potentially more effective than axiomatic oracles in this context, though less broadly applicable and not necessarily correct. Table 3.4 also highlights Jdoctor's worse performance compared to TRATTO, improving mutation scores for 4 out of 6 test suites with implicit oracles (1% average increase vs. Tratto's 7%) and showing no improvement for test suites with regression oracles.

Answer to RQ-5: TRATTO-generated oracles increase the mutation score of test suites with implicit oracles by an average of 7% across six projects, with increases ranging from 2% to 17%. TRATTO also improves mutation scores for test suites with both implicit and regression oracles by an average of 1% across three projects.

3.7 Threats to Validity

Internal Validity

The datasets used in our experiments may contain wrong instances, since they are based on an

automatically generated dataset [8]. We mitigate this threat by manually inspecting and fixing or discarding wrong instances. Manually generated or modified instances were inspected by two authors to minimize bias, and conflicts were solved via open discussion. Indeed, all processes involving manual analysis, including the generation of the ground-truth dataset and the analysis of the predictions by TRATTO, Jdoctor and GPT4, were performed by two authors. All our datasets, results, and tool implementations are available as open source in our replication package [3].

In answering **RQ-1** and **RQ-2**, we measure the differences across the techniques (code models and ablated approaches) in terms of the accuracy in generating the next oracle token. This does not take into account equivalent valid oracles that the model can generate. For example, the oracle "`param < 0;`" is equivalent to "`(param < 0);`", thus both tokens '`(`' and '`param`' are valid when the partial oracle is empty. However, only '`param`' is deemed as correct. The significant difference between the accuracies of the three models in RQ_1 relieves the risk of missing the best LLM for TRATTO. As for **RQ-2**, we noticed that this phenomenon occurred in both cases (for TRATTO and the ablated approaches), partially alleviating this threat to the validity of the results. In **RQ-3** and **RQ-4** we manually analyzed all oracles generated to fully neutralize this threat and properly compare TRATTO against the state-of-the-art approaches, while **RQ-5** is not affected by this threat.

In generating the ground-truth dataset, we decided to keep only those methods featuring at least one oracle, otherwise the dataset would be extremely unbalanced, containing 389 positive samples (oracles) and 6,485 negative samples (units for which no oracles can be extracted). Furthermore, this would make the manual analysis extremely costly, due to the myriad of false positives generated by GPT4. We did analyze the results for TRATTO and Jdoctor considering also the 6,485 negative samples and found that the recall for both remained in similar levels (49% for TRATTO and 19% for Jdoctor) while precision decreased (49% for TRATTO and 57% for Jdoctor). This was expected due to the higher amount of negative samples, making both approaches incur in more false positives.

External Validity

Our experiments with Java methods do not prove the generalizability of TRATTO to other programming languages. Nevertheless, the approach can be easily adapted to other languages since it merely relies on source code and documentation of inputs and outputs. The symbolic component simply needs a language grammar and context restrictions, while for the neural module an ML model pre-trained on the target programming language is sufficient.

TRATTO takes about five times more to analyze a Java class and generate oracles for it, compared to Jdoctor. This is partly because TRATTO generates more oracles. Generating oracles token by token causes an overhead, but acceptable for the improvement achieved. Scalability can be addressed as TRATTO's neural module is decoupled from the symbolic module, and the communication is handled through a REST API [58, 22]. In practice, this means that the neural module could be deployed in a more powerful server, leveraging a bigger code model, to speed up oracle generation.

Chapter 4

LLMs for Generating Concrete Oracles: An Unbiased Large-Scale Study

This chapter presents the findings of a large-scale empirical study designed to evaluate the effectiveness of LLMs in generating concrete test oracles on an unbiased dataset using a reusable methodology. The study investigates how prompt, model type, and size affect the performance of purely neural methods in producing accurate and reliable test oracles. This analysis serves as a foundational step toward identifying the most effective prompt–model combinations for fine-tuning a purely neural technique or the neural component of a broader neural-symbolic approach to infer concrete test oracles. The core contributions of this chapter are the main content of a paper that we presented in the research track of the 2025 International Conference on Automated Software Engineering (ASE'25) [50]

Our empirical evaluation confirms the effectiveness of the neuro-symbolic approach implemented in TRATTO in generating semantically relevant test oracles, and significantly reducing the number of false positives, a main limitation of current state-of-the-art neural approaches [19, 31, 41]. TRATTO generates axiomatic test oracles, which are highly generalizable and independent from the input values, and thus useful for a variety of tasks, including test oracles, program comprehension, requirements specification, and runtime verification. The results of our study on the impact of automatically generated axiomatic oracles on the mutation score of test suites indicate that the wide scope and applicability of axiomatic oracles limits the precision of the generated oracles, and suggest that generating concrete oracles may improve the precision of the generated oracles.

Axiomatic oracles define general properties, for instance, the sum of two positive numbers is a positive number, and sometime miss the precision needed for effectively detecting bugs, for instance a wrong albeit positive sum of two positive integers. Concrete oracles specify the expected values for concrete inputs, and reduce the possibility of false negatives by narrowing the acceptable outcome space from broad intervals to precise values. For practical bug detection and mutation testing, the precision advantage of concrete oracles often outweighs the generalization capabilities of axiomatic approaches.

The availability of training data represents another argument in favor of concrete oracles. The generation of high-quality axiomatic oracle datasets remains constrained to the restricted

collections of existing symbolic tools such as Daikon [20], JDoctor [8], and AutoInfer [73] (among others). These tools inevitably transfer their limitations and biases on the resulting datasets, potentially constraining neural models to inherit their flaws rather than leveraging their full generative potential, creating performance plateaus that reduce the effectiveness of the neural-based approaches. Concrete oracles benefit from virtually unlimited training resources available through the extensive ecosystem of open-source software repositories. High-quality test suites from diverse projects on platforms, like GitHub [26], provide rich and heterogeneous datasets that enable neural models to learn robust patterns without suffering from tool-specific biases. This abundance of naturally occurring test oracles allows for more effective model training and better generalization capabilities across different software domains and testing contexts.

Building on the experience gained through the implementation of TRATTO, developing effective neuro-symbolic approaches for concrete oracle generation demands a deep understanding of the capabilities and limitations of the underlying neural components. Before integrating advanced symbolic constraints and token-level generation strategies, we had to systematically address key questions related to model selection, prompt engineering, and contextual information requirements. The choice of base model can significantly impacts the performance of neuro-symbolic systems. Different model families (such as Llama, Phi, Qwen), specializations (for instance, base, instruct, code) and sizes (parameter counts) exhibit varying capabilities for code comprehension and generation tasks. Understanding how the neural module performs as these parameters change enables informed architectural decisions for subsequent neuro-symbolic implementations, ensuring that symbolic constraints complement rather than hinder neural capabilities.

Prompt engineering represents another critical dimension that requires empirical investigation. The amount and type of contextual information provided to the models–including test prefixes, method signatures, class definitions, and documentation–directly influence the quality of the generated oracles. A systematic evaluation of alternative prompt configurations clarifies the minimal informational content required for effective concrete oracle generation, while revealing potential overload conditions that can impair model performance.

Dinella et al.'s, Hassain et al.'s and Kandaker et al's empirical studies [19, 31, 41] confirm the potentiality of concrete test oracles in revealing bugs. These works evaluates the capability of vanilla LLMs [41] and fine-tuned LLMs [19, 31] on limited-sized public benchmarks, such as Defects4J [40, 77, 74], that are likely included in LLM training data, raising serious concerns about data leakage and inflated performance estimates [1]. Wang et al.s recent survey [70] points out the absence of large-scale evaluation to assess the effectiveness of LLMs in generating assertions for test cases added after the LLM's training cut-off. This highlights a critical gap in understanding the true generalization ability of LLMs for generating oracles [37].

Our empirical study fills this gap through a large-scale evaluation of vanilla LLMs for concrete test oracle generation. The study employs an unbiased dataset of 13,866 test oracles extracted from 135 open-source Java projects, with all test cases created after 2024-09-01, to ensure full separation from model training data and eliminate the data leakage issues that undermined prior evaluations.

The evaluation methodology encompasses two complementary analyses: a correctness evaluation, obtained through comparison with programmer-written oracles, and an effectiveness evaluation, determined by the contribution of the generated oracles to mutation score improvement. Together, these analyses provide both semantic validation and empirical evidence of practical utility, ensuring that the evaluation outcomes generalize to real-world testing envi-

ronments.

The empirical study reveals several counterintuitive findings that challenge conventional assumptions about LLMs application to code generation tasks. As expected, larger models consistently outperform smaller variants, but code-specialized models show no significant advantage over general-purpose alternatives. Most surprisingly, additional contextual information beyond test prefixes and method calls fails to improve oracle generation quality, suggesting that effective concrete oracles can be generated with minimal context requirements.

The mutation score analysis shows that LLM-generated oracles achieve performance comparable to human-written oracles (43% vs. 45% average mutation score in our experiments), indicating substantial practical utility despite syntactic differences from programmer-written assertions. However, the study reveals specific limitations that constrain the effectiveness of purely neural approaches. The mutation analysis shows that compilation failures and test failures affect a notable portion of generated oracles, requiring filtering 32 compilation failures and 8 test failures out of 79 oracles tested across 4 projects, in order to create compilable green test suites. These findings establish the viability of neural approaches for concrete oracle generation while highlighting specific technical shortcomings that motivate the development of hybrid neuro-symbolic solutions discussed in Chapter 5.

The remainder of this chapter systematically investigates the capabilities of vanilla LLMs for concrete oracle generation through 4 research questions.

Section 4.1 presents the empirical study methodology and experimental setup, including dataset construction, prompt design, and evaluation metrics.

Section 4.2 reports the experimental results for oracle correctness analysis, addressing three complementary questions: (i) how different LLM fundamental properties (model family, size, and specialization) affect generation accuracy (**RQ-1**), (ii) whether the amount and type of contextual information provided in prompts influences oracle quality (**RQ-2**), and (iii) which types of assertion methods are more effectively generated by LLMs (**RQ-3**).

Section 4.3 evaluates the robustness of LLM-generated oracles through mutation testing analysis, investigating whether the generated oracles can effectively detect code modifications and maintain test suite effectiveness (**RQ-4**).

Section 4.4 discusses the internal and external threats to validity that may affect the credibility and generalizability of the results, and outlines the measures adopted during the study to mitigate these risks.

4.1 Methodology

Figure 4.1 presents an overview of our empirical study methodology, which centers around a dataset of 13,866 test oracles that we extracted from GitHub repositories (Section 4.1.1) and evaluates using 4 prompt configurations (Section 4.1.2) across 10 different LLMs (Section 4.1.3) through correctness metrics (Section 4.2) and mutation analysis (Section 4.3).

Table 4.1 summarizes the scale and configuration parameters of our empirical investigation.

4.1.1 Dataset

We constructed an unbiased dataset of test cases by implementing a systematic approach that first selects candidate repositories and then extracts recent test cases to ensure temporal sepa-

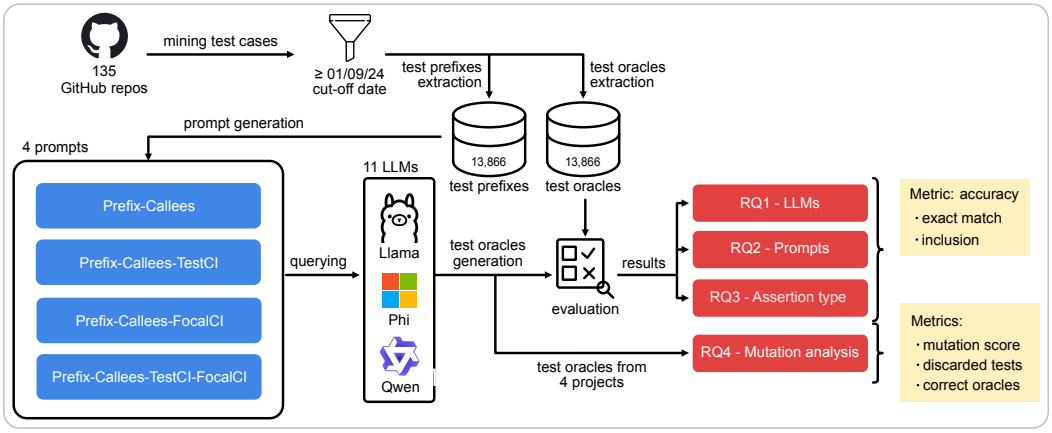

Figure 4.1. Overview of our methodology for the empirical study of concrete oracle generation.

ration from LLM training data.

Repository Selection Criteria

We identified public and non-fork repositories from GitHub using SEART GitHub Search [17] with the following quality filters [47]:

- At least 100 commits
- At least 50 issues
- At least 10 distinct contributors
- At least 10 stars
- At least 1 commit between 2024-09-01 and 2025-05-05

This selection process yielded 4517 candidate repositories. To facilitate the subsequent mutation analysis, we retained only repositories containing a single `pom.xml` file that successfully compiles with Maven, resulting in 135 Maven projects. We did not verify that test cases pass during this initial filtering, as we later perform mutation analysis exclusively on passing test suites.

Test Case Extraction Process

We constructed an unbiased repository of programmer-written test cases by extracting test cases introduced after 2024-09-01, according to the commit history of the projects. This temporal cutoff ensures complete separation from LLM training data, at the time of our experiments, eliminating potential data leakage concerns.

We consider a test oracle any occurrence of *assert* statement within a test case. The test prefix of an oracle encompasses all code in the test case preceding the given oracle, including any prior assertions. We obtained 13,866 test oracles with their corresponding prefixes, all manually created by project developers after the specified cutoff date.

Table 4.1. Corpus and configuration of the empirical study.

Candidate repositories	4517
Repositories that compile with Maven	135
Manually designed test oracles	13,866
Time window of mined commits	2024-09-01 to 2025-05-05
Evaluated LLMs	10
Prompt strategies per oracle	4
Concrete prompts (#oracles × #strategies)	55,464
Total LLM queries (#concrete prompts × #LLMs)	554,640

Algorithm 1 at page 57 outlines the procedure that we defined to identify the test cases created after the start date and retrieve their final versions. The algorithm processes each commit generated after the specified date in chronological order, identifies the modified test files, analyzes the source code of both new and old versions to extract test cases, adds new test cases to the target set, removes any previously added test cases that were subsequently deleted, and updates the body of any test case that was modified in subsequent commits.

The algorithm identifies the test files that have been modified (line 1.4) for each commit after a given date in chronological order (lines 1.3–1.16), analyzes the source code of the new and old versions of the test file to extract the test cases defined in both versions (if the file is added in the current commit, the set of test cases in the old version is empty), adds the new test cases to the target set T_{new} (lines 1.6–1.7), removes from T_{new} any test case added in a previous commit and removed in the current one, and updates the body of any test case added in a previous commit and modified in the current one (line 1.13) based on fully identical definitions. It returns the test cases collected (line 1.17).

Algorithm 2 at page 58 shows how we construct the oracle dataset from the extracted test cases. The algorithm builds a dataset where each element represents a 5-tuple containing the test class, test prefix, focal class, invoked methods, and target assertion.

The *prefix* contains all statements preceding the target assertion, including any prior assertions. The *focal class* represents the class containing the methods that the test case tests. A test case might call other methods (including methods from other classes) as well, in the prefix, to build and prepare objects or set up a state for the call to the method under test. Our dataset contains unit test cases. In a unit test, the method under test (the focal method) is typically called immediately before the assert statement, but it also might be called earlier or called in the assertion (Listing 4.1). Our methodology does not attempt to determine what the focal method is. Instead, it leverages the standard unit testing convention where test class *MyClassTest* tests the methods of source code class *MyClass* (algorithm 2, line 2.4).

The state-of-the-art approaches heuristically assume that the focal method is the last method called in the test case [19, 31, 41]. This assumption does not always hold. For example, the test case `test12()` in Listing 4.1 tests method `shift` called before the `assertEquals` statement, and before the call to method `length`, a getter method that `test12()` calls at the end to check the post-condition of method `shift`. The state-of the-art approaches wrongly consider method `length` and not method `shift` as the focal method. Our methodology treats fairly all method calls in the test case, and collects the definitions of all the constructors and methods invoked within the test prefix.

A test case can contribute more than one oracle if it contains more than one assertion. In this case, the number of data-points obtained by processing the original test case is equal to $|A|$. The final dataset contains test cases with 1 (34%), 2 (17%), 3 (11%), and 4 or more (38%) assertions. When the test cases contains multiple assertions, the prefix of each assertion includes the assertions that occur above in the test case. The many test cases with single assertions in the dataset make our study applicable for generating assertions for test prefixes without assertions, for instance, for automatically generated test cases, like EvoSuite test cases without regression oracles. The many test cases with multiple assertions in the dataset make our study applicable also for adding assertions to test cases that already include some assertions, like manually generated test cases.

```
// Original Test case
public void test12()  throws Throwable  {
      BinarySignal binarySignal0 = new BinarySignal(32767);
      BinarySignal binarySignal1 = binarySignal0.shift((int) (byte)0);
      assertEquals(32767, binarySignal1.length());
}
// Test prefix
public void test12()  throws Throwable  {
      BinarySignal binarySignal0 = new BinarySignal(32767);
      BinarySignal binarySignal1 = binarySignal0.shift((int) (byte)0);
}
// Focal method
public final int length() {
    return length;
}
// Target assertion
assertEquals(32767, binarySignal1.length());
```

Listing 4.1. A test case with one assertion, from the TOGLL replication package [31].

4.1.2 Prompt Design

We designed 4 prompt configurations with progressively increasing content to investigate the impact of contextual information on oracle generation quality:

- **Prefix-Callees** (**P-C**): the test prefix and source code of methods called within the test case
- **Prefix-Callees-TestCl** (**P-C-T**): the **P-C** and the Javadoc and definition of the fields and the Javadoc, signature, and body of the methods defined in the test class
- **Prefix-Callees-FocalCl** (**P-C-F**): the **P-C** and the field definitions and methods from the focal class
- **Prefix-Callees-TestCl-FocalCl** (**P-C-T-F**): all available contextual information from both test and focal classes (the union of the former prompts)

Algorithm 3 details the process that generates prompts from predefined templates for each oracle in the dataset, resulting in 55,464 concrete instances used in our experiments (4 prompt strategies $\times$ 13,866 oracles).

The algorithm extracts the elements required to build the prompt (prefix, signature, Javadoc and body of the methods invoked in the test prefix, and fields and methods defined within the focal class and the test class [18], depending on the prompt type) and initializes an empty list `prompts` of prompts (lines 3.1–3.2).

The algorithm iterates over all the oracles in the input dataset D, and generates the prompt specified in the input template for each of oracle (lines 3.3–3.15).

At each iteration, the algorithm initializes the prompt with the *test prefix* and the information about the methods invoked within it (lines 3.6–3.8). It adds the information required for the focal (lines 3.9–3.11) and test class (lines 3.12–3.14) with procedure AddClassInfo, as indicated in the prompt type.

The configuration file provided to the algorithm (*pspec*) contains the types of information to be included to progressively compose the prompt with the signature, the Javadoc and the body of the methods invoked in the test prefix, and eventually the fields and the methods defined within the focal class and the test class (line 3.1). The procedure iterates over the **13,866** oracles data points collected in the previous phase, integrating the pieces of information within the prompt (lines 3.3–3.15). The algorithm enriches the prompt with a clear description of the task that each model must accomplish and generates as output a list of **13,866** final prompts (line 3.17).

We fit as much information as possible in the prompts while respecting the maximum window context length of the model, leveraging an algorithm that implements the E-Wash approach [14].

We built the 4 prompts for each of the 13,866 oracles in our dataset, for a total of 55,464 prompts that we used to query the LLMs to generate a test oracle without any example (*zero-shot* learning).

4.1.3 Large Language Models

To study how *model size* and *training specialization* impact oracle prediction, we selected 10 different LLMs that use different training and inference techniques: **General**, **Reasoning**, **Instruct**, and **Code**, and size ranging from **1B** to **70B** parameters.

General-purpose models trained on web-scale mixed-domain datasets:

- **Qwen2.5** (1.5B, 14B, and 32B)
- **Phi4-Mini** (3.8B), **Phi4** (14B)

Reasoning-enhanced models with explicit chain-of-thought analytical training:

- **Phi4-Reasoning** (14B)

Instruction-tuned models fine-tuned with human-based feedback or datasets specialized in natural-language instructions:

- **Llama3.3** (70B)

Code-specialized models trained or fine-tuned on source code datasets and expected to excel at syntactic and semantic understanding typical of coding tasks:

- **Qwen2.5-Coder** (1.5B, 14B, 32B)

This selection enables systematic comparison of model families with multiple sizes (for instance, **Qwen2.5** at 1.5B, 14B, and 32B parameters) to isolate the effects of scale from those of training data specialization. We evaluate whether oracle generation benefits more from scale or task-oriented pre-training, by comparing domain-specialized models (Code and Reasoning) with their general-purpose counterparts.

4.2 Experimental Results

The experimental evaluation investigates three complementary research questions about the accuracy of LLM-generated oracles:

RQ1 ***Impact of the LLM****: Does the choice of LLM affect the accuracy of the generated oracles?* We compare different LLMs to evaluate the influence of the type and size of the LLM on the generation process.

RQ2 ***Impact of the Prompt****: Does the choice of prompt affect the accuracy of the generated oracles?* We evaluate different input information to study the tradeoff between the amount of information provided with the prompt and the effectiveness of the LLM.

RQ3 ***Impact of the Type of Oracle****: Are some assertion methods easier for LLMs to accurately generate?* We compare the accuracy of the LLM output for different `assert*` methods.

The evaluation employs two complementary metrics: the *exact match* accuracy, where generated oracles must be character-for-character identical to programmer-written oracles, and *inclusion* accuracy, where this assumption is relaxed and the original oracle constitutes a substring of the generated one (LLMs do not always perfectly follow the instructions provided, failing to format the response as desired or generating more than one oracle). String matching and substrings can be easily automated, which is essential for a large-scale experiment. Exact matching misses generated oracles that semantically match the original oracles, while expressed in a different syntactic form. We observed that exact matches sometimes failed due to small and easily fixable syntactic differences, like a missing semicolon.

Our experiments generate oracles, by systematically removing developer's oracles from test cases, prompting LLMs to generate the target oracles for the test prefixes, and comparing the LLM-generated outputs with the original developer's assertions. This methodology uses the 13,866 manually designed oracles from 135 projects as our evaluation baseline.

The comprehensive evaluation encompasses 10 LLMs queried with 4 different prompt configurations for each oracle, generating a total of 554,640 predictions. We acknowledge that both exact matching and inclusion matching represent pessimistic under-approximations of LLM accuracy, as they miss semantically equivalent oracles expressed through different syntactic forms.

4.2.1 RQ-1: Impact of LLM on the Generated Oracles

Table 4.2 presents the comprehensive accuracy results for each combination of LLMs and prompt configurations, with models sorted by average performance. In any row, the *no oracle* column indicates the percentage of LLM outputs that fail to contain assert statements, representing a generation failure, the *exact match* column is a subset of the *inclusion* column, and the *inclusion* column plus the *no oracle* column is less than or equal to 100%.

LLM			Prompt	Accuracy						Time
Type	Model	Size	Type	exact match		inclusion				(s)
				#	%	#	%	avg.	no oracle	
Code	Qwen2.5-Coder	32b	P-C	806	5.8%	3,993	28.8%	29.4%	8.4%	1.04
			P-C-T	673	4.8%	4,067	29.3%		8.0%	4.05
			P-C-F	821	5.9%	4,010	28.9%		7.2%	1.16
			P-C-T-F	621	4.5%	**4,266**	**30.7%**		7.4%	4.17
General	Qwen2.5	32b	P-C	2,923	21.0%	3,960	28.5%	29.2%	5.0%	1.06
			P-C-T	2,883	20.8%	4,100	29.5%		4.2%	4.08
			P-C-F	2,944	21.2%	4,026	29.0%		5.0%	1.18
			P-C-T-F	**3,762**	**27.1%**	4,124	29.7%		3.9%	4.28
Instruct	Llama3.3	70b	P-C	1,379	9.7%	3,118	21.9%	22.9%	3.9%	N/A
			P-C-T	1,714	10.3%	3,868	23.2%		1.4%	N/A
			P-C-F	1,463	10.5%	3,146	22.7%		1.5%	N/A
			P-C-T-F	1,911	11.4%	4,015	24.0%		1.2%	N/A
Code	Qwen2.5-Coder	14b	P-C	968	7.0%	2,946	21.2%	20.9%	1.6%	0.58
			P-C-T	889	6.4%	2,790	20.1%		3.4%	2.18
			P-C-F	939	6.8%	2,976	21.4%		2.0%	0.64
			P-C-T-F	943	6.8%	2,870	20.7%		3.0%	2.28
General	Qwen2.5	14b	P-C	233	1.7%	2,910	21.0%	20.7%	1.8%	0.59
			P-C-T	337	2.4%	2,866	20.6%		3.2%	2.18
			P-C-F	362	2.6%	2,886	20.8%		1.7%	0.66
			P-C-T-F	316	2.3%	2,823	20.3%		2.5%	2.29
General	Phi4	14b	P-C	1,562	11.2%	2,566	18.5%	17.8%	0.8%	0.78
			P-C-T	1,429	10.3%	2,424	17.5%		3.8%	2.69
			P-C-F	1,515	10.9%	2,464	17.7%		1.0%	0.91
			P-C-T-F	1,426	10.3%	2,398	17.3%		3.7%	3.36
General	Phi4-Mini	3.8b	P-C	560	0.1%	642	4.6%	4.6%	0.9%	0.60
			P-C-T	13	0.1%	634	4.6%		2.9%	1.42
			P-C-F	28	0.2%	605	4.4%		0.9%	0.76
			P-C-T-F	14	0.1%	636	4.6%		2.1%	1.44
Reasoning	Phi4-Reasoning	14b	P-C	16	0.1%	642	4.6%	4.6%	77.3%	38.47
			P-C-T	13	0.1%	634	4.6%		76.3%	42.11
			P-C-F	28	0.2%	605	4.4%		78.3%	36.17
			P-C-T-F	14	0.1%	636	4.6%		75.3%	41.99
General	Qwen2.5	1.5b	P-C	148	1.1%	536	3.9%	3.3%	3.6%	0.24
			P-C-T	39	0.3%	403	2.9%		7.2%	0.61
			P-C-F	90	0.6%	486	3.5%		5.4%	0.27
			P-C-T-F	54	0.4%	395	2.8%		6.5%	0.62
Code	Qwen2.5-Coder	1.5b	P-C	155	1.1%	436	3.1%	2.7%	12.6%	0.21
			P-C-T	120	0.9%	339	2.4%		17.6%	0.56
			P-C-F	147	1.1%	410	3.0%		14.8%	0.22
			P-C-T-F	126	0.9%	329	2.4%		15.3%	0.58

Table 4.2. Accuracy for each combination of LLMs and prompts. The models are sorted by the "average" column.

Model Size Effects

The experimental results demonstrate a pronounced correlation between the number of parameters of the model and the oracle generation accuracy. The largest models consistently achieve superior performance: Qwen2.5-32b, Qwen2.5-Coder-32b, Llama3.3-70b, Qwen2.5-Coder-14b, Qwen2.5-14b, Phi4-14b. This scaling effect proves remarkably consistent, with inclusion accuracy improving approximately ten-fold when scaling from 1.5B to 32B parameters (from 3% to 29%). The only notable exception occurs with Phi4-Reasoning-14B, which performs poorly despite its substantial parameter count, generating no oracle output over 75% of the time. This anomaly suggests that specialized reasoning training may interfere with concrete oracle generation capabilities.

Model Type Analysis

Surprisingly, code-specialized models demonstrate no significant advantage over general-purpose alternatives. For instance, Qwen2.5-Coder exhibits nearly identical performance to Qwen2.5 across all parameter scales. This counterintuitive finding challenges the assumption that domain-specific training necessarily improves code generation tasks.

We hypothesize that Qwen2.5's superior performance stems from its training dataset composition, which emphasizes code, mathematics, and technical knowledge. The technical report indicates that pre-training data expanded from 7 trillion to 18 trillion tokens with explicit focus on coding content, potentially explaining why the general model matches code-specialized variants [75].

Answer to RQ-1: The scale of the model is the main factor determining the accuracy of oracle generation, whereas training specialization has minimal impact. Larger general-purpose models are preferable to smaller code-specialized ones when resources are constrained.

4.2.2 RQ-2: Impact of Prompt Information Content

The analysis of prompt configurations yields surprising findings about the utility of contextual information. Although the amount of information increases progressively across the four prompt types, from basic *Prefix-Callees* (*P-C*) to comprehensive *Prefix-Callees-TestCl-FocalCl* (*P-C-T-F*), the resulting accuracy gains remain minimal across all models.

Information Content Analysis

Table 4.2 shows that incorporating focal and test class information into the basic prompt configuration changes the accuracy by less than one percentage point for most models. Such marginal gains do not justify the considerable increases in context length, token usage, and inference latency introduced by richer prompts.

Context Window Implications

The results suggest that test prefixes and called method signatures provide sufficient context for effective oracle generation. Additional structural information from surrounding classes contributes little to LLM understanding of expected oracle behavior, despite containing potentially relevant semantic information.

Answer to RQ-2: Minimal prompts containing test prefixes and invoked method definitions suffice for optimal oracle generation. Additional contextual information increases computational costs without corresponding accuracy benefits, making concise prompts preferable for practical deployment.

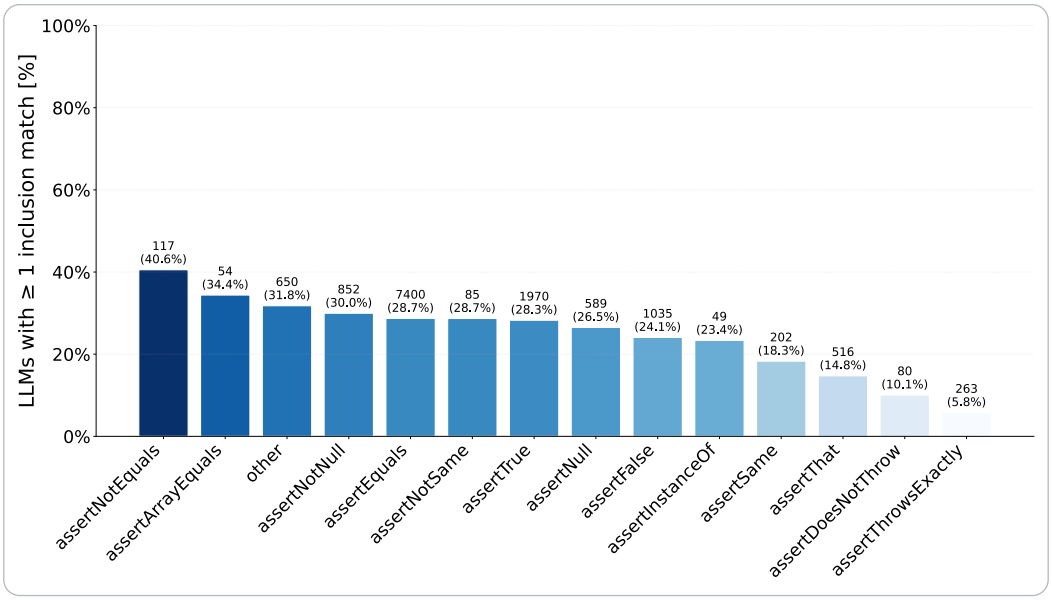

Figure 4.2. For each assertion type or method, how often at least one of the four prompts of each LLM exactly matched the human-written oracle.

4.2.3 RQ-3: Oracle Type Difficulty Analysis

Figure 4.2 illustrates the proportion of generated oracles that match the developer-written ones across different assertion types, based on inclusion matching. The analysis groups oracles by their underlying `assert*` method calls, consolidating infrequent types (fewer than 10 instances) into an *Other* category.

Assertion-Type Performance

Oracle generation performance varies across assertion types, typically ranging from 20% to 40%. `assertNotEquals` and `assertArrayEquals` stand out with 40% and 34% accuracy, respectively, whereas many other types remain below 20%.

The diagram reports the oracle type on the x-axis, and labels the bars with the percentage and the number of oracles of the given type. The bars are labeled with the total number of oracles and the percentage of matching oracles for each type. The different distribution among the assertion types for nine of the types varies from 23% (`assertInstanceOf`) to 34% (`assertArrayEquals`), with a percentage over 40% for `assertNotEquals`, and a percentage below 20% for four types of oracles. The analysis shows that the type of oracle has some impact on the generation process, with some outliers performing both better and worse than the

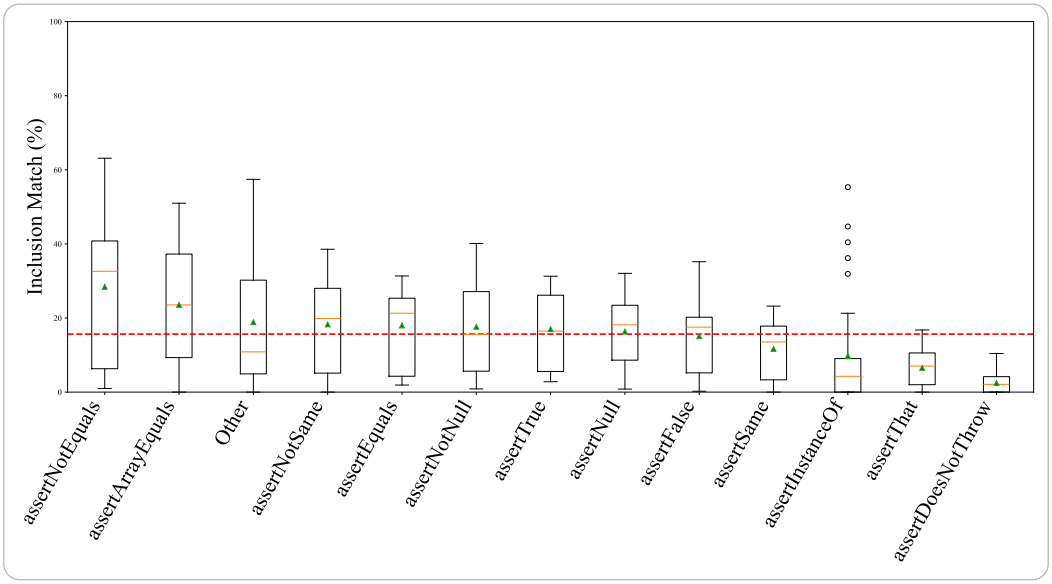

Figure 4.3. Average of all the 554,640 oracle predictions per type of assertion (the dash line represents the overall mean).

majority of types. However, Figure 4.2 refers to the inclusion matching, which does not capture semantic matches that differ syntactically, and this does not allow us to generalize the result.

Figure 4.3 provides complementary analysis through box plots summarizing all 554,640 oracle predictions across assertion types. The consistent patterns between both visualizations confirm that no specific LLM-prompt configuration consistently excels at particular assertion types.

Statistical Analysis

Pearson correlation analysis between assertion type frequency and matching rates yields $r = 0.112$, $p = 0.72$, confirming no significant relationship between oracle type prevalence and generation accuracy. This finding suggests that the difficulty in generating assertion oracles is driven by semantic rather than syntactic factors.

Answer to RQ-3: While oracle type influences generation effectiveness, the differences remain modest and inconsistent across configurations. The inclusion matching methodology limits our ability to draw definitive conclusions about semantic generation capabilities across different assertion categories.

4.3 Mutation Testing Analysis

The second part of the evaluation investigates the practical effectiveness of the generated oracles in realistic testing scenarios, addressing the following research question:

RQ4 ***Effectiveness of the Oracles****: Do the generated oracles enhance the mutant detection capability of test cases?* We compare mutation scores across four distinct configurations to isolate and quantify the impact of generated oracles on overall test suite effectiveness.

The research question investigates whether LLM-generated assertions enhance the fault-detection capability of test suites. This analysis evaluates robustness through mutation testing, quantifying how effectively the generated oracles can detect behavioral deviations introduced by code mutations.

4.3.1 RQ-4 Oracle Effectiveness Through Mutation Analysis

The *mutation* analysis employs the best-performing configuration identified in the *Oracle Correctness Analysis* 4.2: *Qwen2.5-32B* with the *Prefix-Callees-TestCl-FocalCl* (*P-C-T-F*) prompt configuration. The experiment evaluates a sample set of four projects that meet the requirements of compilation, execution and PIT [15] support.

The analysis measures mutation scores across four scenarios:

- **Original Oracles**: Test cases with developer-written assertions
- **Without oracles**: Test prefixes with all assertions removed
- **Generated oracles**: Test cases augmented exclusively with LLM-generated assertions
- **Combined oracles**: Test cases containing both original and generated oracles

These four configurations allow us to isolate the effect of generated oracles and assess their contribution both independently and in conjunction with existing ones.

Algorithm 4 outlines the process to compute the mutation scores for the classes of a repository. The algorithm initializes the suite of green test classes to an empty set (line 4.1) and iterates over the list of test classes in the dataset D of oracle datapoints to verify that their tests compile and pass, otherwise it discard them (lines 4.2–4.8). The algorithm adds the refined test classes to the suite (line 4.11), and computes the mutation score of the tests containing the original assertions (lines 4.2–4.12). The algorithm removes the original test oracles written by the developers from the test cases of the test classes in the suite (lines 4.14–4.18) and computes the mutation score of the tests without oracles (line 4.19). The algorithm adds the LLM-generated oracles to the test cases without oracles, only if the updated test cases compile and pass, (lines 4.20–4.27) and computes the mutation score of the test cases containing only the generated oracles (line 4.28). Finally, the algorithm includes the original oracles to the test cases containing the LLM-generated oracles (lines 4.29–4.36) and computes the mutation score of the test cases embedding both the original and the generated oracles (line 4.37).

Compilation and Test Execution Results

To compute the mutation score, we had to discard all oracles that fail since PIT requires a green test suite. Table 4.3 reports the number of generated oracles that do not compile (*Compilation Failure*), fail (*Test Failure*), and pass (*Test Passes*) for the four projects we used in this research question. We discarded between 1 in *twilio/twilio-java* (17%), and 33 in *wmixvideo/nfe* (55%) oracles, 10 oracles on average (40%). We manually verified that the 8 oracles that fail (*Test Failure*) are incorrect (false positives). The test cases that fail and that PIT ignores may still be useful to uncover real bugs (for instance, failures due to correct oracles in a faulty program).

We manually verified that the 39 oracles that pass (*Test Passed*) are correct (no false negatives), highlighting the potential of LLMs for creating oracles useful for mutation testing.

Mutation Score Analysis

Table 4.4 presents the results of the mutation testing in the four evaluation scenarios. The findings reveal clear patterns that highlight the effectiveness of LLM-generated oracles. The generated oracles significantly improve the test cases without oracles, producing relative increments ranging from 76% to 443%. For example, in the *twilio-java* project, the mutation score increases from 3% to 15%, while in the *bkiers/liqp* project, it rises from 7% to 38%. These substantial improvements demonstrate that LLM-generated oracles provide meaningful enhancement to test suites without assertions.

When comparing generated oracles to the original developer-written oracles, the results reveal surprisingly competitive performance. The generated oracles achieve mutation scores similar to the original oracles, with a loss of 40% in the worst case and a gain of 30% in the best case, averaging only a 10% reduction. This relatively small performance gap indicates the high quality of the generated oracles. The average mutation score for generated oracles reaches 43%, closely approximating the 45% achieved by original oracles. In the *wmixvideo/nfe* project, generated oracles even outperform the original ones, achieving a 79% mutation score compared to 61% for the developer-written oracles.

The combination of generated oracles with original oracles produces mixed results. Only the *wmixvideo/nfe* project shows improvement when both oracle types are present together, with the mutation score increasing from 61% to 80%. The other three projects maintain the same scores when combining both oracle types, suggesting that generated oracles often target similar vulnerability patterns as developer-written assertions. The significant improvement observed in *wmixvideo/nfe* indicates that when generated oracles achieve high mutation scores, they can effectively complement existing oracles to enhance overall test suite effectiveness.

While the improvement over test cases with no oracles represents an expected outcome, the competitive performance relative to carefully crafted developer-written oracles proves quite surprising and indicates the substantial quality of the generated oracles. The fact that generated oracles can achieve comparable mutation detection capabilities to human-written assertions demonstrates their practical utility for automated testing scenarios.

Manual examination of all 79 LLM-generated oracles across the four projects reveals several important characteristics. Developers create assertions with specific literals that prove hardly predictable by LLMs, such as complex encoded strings like "`%3C%3Fxml+version%3D%221.0%22+encoding%3D%22UTF-8%22%3F%3E%3CConversationRelay%2F%3E`". In contrast, LLMs produce weaker but still-correct assertions that typically contain references to previous elements of the test prefix, such as "elem.getElementAttributes().size()". Despite this difference in specificity, the generated oracles maintain their effectiveness in detecting code modifications through mutation testing.

The analysis also demonstrates that LLMs can extract patterns from test prefixes containing multiple assertions and correctly predict subsequent assertions. However, LLMs occasionally fail to understand the underlying semantics, sometimes due to missing context in the prompt. These limitations appear rooted in information retrieval techniques rather than fundamental model capabilities, suggesting potential improvements through advanced contextual information augmentation approaches. The mutation testing results validate that LLM-generated oracles are effective and go beyond pure syntactic correctness, demonstrating measurable fault

Project	Compilation Failure	Test Failure	Test Passed
twilio/twilio-java	1	0	5
bkiers/liqp	0	2	3
wmixvideo/nfe	29	4	27
wearefrank/ladybug	2	2	4
TOTAL	32	8	39

Table 4.3. Number of test cases that do not compile, fail, and pass with the generated oracles.

Project	**Original oracles**	**Without oracles**	**Generated oracles**	**Original + Generated oracles**
twilio/twilio-java	25%	3%	15%	25%
bkiers/liqp	42%	7%	38%	42%
wmixvideo/nfe	61%	45%	79%	80%
wearefrank/ladybug	52%	19%	41%	52%
Average	45%	19%	43%	50%

Table 4.4. Mutation scores (%) with different test oracles. The test code is identical, and only the oracles differ.

detection capabilities that justify their integration into automated testing workflows.

4.4 Threats to Validity

Internal Validity

The results may vary with different parameters and temperatures. We mitigated the risk of biased results by using only default LLM configurations, reporting the parameters and temperature used in the experiments, and making the benchmark publicly available in a replication package to allow the reproduction of the results.

The results may change with the future LLM milestones. We mitigated the risk of aging results by experimenting with 10 popular LLMs of different sizes, to investigate the impact of the LLM on the generation process. We experimented with open source LLMs only. Proprietary LLMs, like OpenAI's models, may perform better than the LLMs we evaluate in this paper.

The manual validation of the 13,866 generated oracles is almost impossible. We automatically validated the generated oracles by comparing them with the available and carefully designed oracles. We mitigated the risks of incorrect validation by implementing the comparison with string matching and inclusion that misses valid oracles that differ from the reference manual oracles only for even small syntactic elements. We decided to report conservative results that are a pessimistic under-approximation of the actual data, to avoid any risk of overestimating the results.

The test mining and mutation analysis setup may miss some test cases and mutants, thus reducing the effectiveness of the results. We rely on publicly available and well-known tools

(PyDriller, Tree-Sitter, JavaParser, Maven, and PIT) to reduce the risk of mis-minings and mis-mutations due to errors in the tools.

External Validity

The results may be biased by the data used to build the benchmark and the building process.

We mitigated the risk of biased benchmarks by mining Java/Maven projects from popular repositories, and limiting the subjects to test cases added after September 2024, the last reported training cutoff date of the LLMs we used.

We experimented only with LLMs with a publicly declared cutoff date, and we executed all LLMs locally. We relied only on open-source LLMs for reproducibility. We excluded OpenAI models that are only available on the cloud and do not declare the cutoff date, to avoid the risk of biases due to executions out of our control.

Algorithm 1 Extracting test cases created after a given date.

Input: repository r, starting date `since`
Output: test cases T_{new} created after *since*

$commits \leftarrow$ PyDriller(r, `since`) ▷ date-ordered
$T_{\text{new}} \leftarrow \emptyset$ ▷ recent tests; the output of this procedure
for $c \in commits$ (main branch) **do**
  for $f \in c$.modified_files **do**
    if IsJavaTestClass(f) **then**
      $code_{\text{before}} \leftarrow$ SourceCodeBefore(c, f)
      $code_{\text{after}} \leftarrow$ SourceCodeAfter(c, f)
      $T_{\text{before}} \leftarrow$ GetTestMethods($code_{\text{before}}$)
      $T_{\text{after}} \leftarrow$ GetTestMethods($code_{\text{after}}$)
      $T_{\text{add}} \leftarrow T_{\text{after}} \setminus T_{\text{before}}$
      $T_{\text{del}} \leftarrow T_{\text{before}} \setminus T_{\text{after}}$
      $T_{\text{mod}} \leftarrow$ ExtractTestDiff($T_{\text{after}}, T_{\text{before}}$) ▷ T_{add}, T_{mod}, and T_{del} are disjoint
      $T_{\text{new}} \leftarrow (T_{\text{new}} \cup T_{\text{add}} \cup T_{\text{mod}}) \setminus T_{\text{del}}$
    end if
  end for
end for
return T_{new}
procedure GetTestMethods($code$)
  if $code =$ `None` **then**
    return $\emptyset$
  end if
  $tree \leftarrow$ TreeSitterParse($code$)
  return $\{m \in tree \mid m$ is a test method$\}$
end procedure
procedure ExtractTestDiff($T_{\text{after}}, T_{\text{before}}$)
  $T_{\text{mod}} \leftarrow \emptyset$
  for all $t_a \in T_{\text{after}}$ **do**
    if t_a added after `since` **then**
      $t_b \leftarrow$ Find($T_{\text{before}}, \lambda t.t.\text{sig} = t_a.\text{sig}$)
      if $t_b \neq$ `None` **then**
        if t_a.body $\neq t_b$.body **then**
          $T_{\text{mod}} \leftarrow T_{\text{mod}} \cup \{t_a\}$
        end if
      end if
    end if
  end for
  return T_{mod}
end procedure

Algorithm 2 Building dataset D of oracles.

Input: Repository r, test cases T from algorithm 1

Output: Dataset D of oracles, where each element is a 5-tuple $\langle tc, prefix, fc, invoked, a \rangle$, with test class tc, test prefix $prefix$, focal class fc, $invoked$ as the methods called in the prefix, and a the next assertion to predict.

$D \leftarrow \emptyset$ ▷ Dataset of oracles (output of this function)
$S \leftarrow \emptyset$ ▷ Set of ⟨ focal class, test class ⟩ pairs
for each fc in SOURCECLASSES(r) **do**
    $tc\_name \leftarrow fc.name +$ `"Test"` ▷ Test method name
    $tc \leftarrow$ GETTESTCASE(tc_name)
    if $tc \neq$ **None then**
        $S \leftarrow S \cup \langle fc, tc \rangle$
    end if
end for
for each $\langle fc, tc \rangle \in S$ **do**
    for each test t in $tc.tests$ **do**
        $t.body \leftarrow$ ENRICHTESTCASE($t.body$)
        $A \leftarrow$ EXTRACTASSERTIONS($t.body$)
        for each $a \in A$ **do**
            $prefix \leftarrow t.body.stmts[0..a]$ ▷ All statements (and prior assertions, if any) before a
            $invoked \leftarrow \emptyset$
            for each method call mc in $t.body$ **do**
                $invoked \leftarrow invoked \cup mc.decl \cup mc.decl.body$
            end for
            $D \leftarrow D \cup \{\langle tc, prefix, fc, invoked, a \rangle\}$
        end for
    end for
end for
return D
procedure ENRICHTESTCASE(body) ▷ Embed bodies of methods containing assertions called within the test case
    for each method call mc in $body$ **do**
        if ISMETHODWITHASSERTIONS(mc) **then**
            EMBEDBODY($body$, $mc.decl.body$) ▷ Integrate statements and assertions of the method, refactoring variables for consistency
        end if
    end for
    return $body$
end procedure

Algorithm 3 Building prompts.

Input: Oracle dataset D from algorithm 2, and a prompt specifier `pspec` that contains two Boolean fields `includeFocalClass` and `includeTestClass`.

Output: One prompt for each oracle in the dataset, as specified in the prompt type.

```
1:  procedure BuildPrompt(pspec)
2:      prompts ← {}
3:      for ⟨tc, prefix, fc, _, _⟩ ∈ D do
4:          p ← prefix
5:          ims ← tc.invoked
6:          for m ∈ ims do
7:              p ← p ‖ m
8:          end for
9:          if pspec.includeFocalClass then
10:             AddClassInfo(p, fc)
11:         end if
12:         if pspec.includeTestClass then
13:             AddClassInfo(p, tc)
14:         end if
15:         prompts ← prompts ∪ {p}
16:     end for
17:     return prompts
18: end procedure

19: procedure AddClassInfo(p, c)
20:     for f ∈ c.fields do
21:         p ← p ‖ f.signature
22:     end for
23:     for m ∈ c.methods do
24:         p ← p ‖ m
25:     end for
26: end procedure
```

Algorithm 4 Mutation analysis for generated oracles.

Input: repository `r`, oracle dataset `D` from Algorithm 2
Output: Mutation score with and without the LLM-generated assertions

```
1: green_suite ← ϵ
2: for each ⟨ tc, _, _, _, _ ⟩ in D do
3:     green_suite ← DUPLICATE(tc)
4:     green_suite.body ← ϵ
5:     for each t in tc.tests do
6:         green_suite.body ← green_suite.body ∪ t
7:         if PASSINGTESTS(r, tc) == False then
8:             green_suite.body ← green_suite.body \ t
9:         end if
10:    end for
11:    green_suite ← green_suite ∪ green_suite
12: end for
13: score_orig ← RUNPIT(r, green_suite)
14: for each tc in green_suite do
15:    for each t in tc.tests do
16:        REMOVEORIGINALASSERTION(t)
17:    end for
18: end for
19: score_without ← RUNPIT(r, green_suite)
20: for each tc in green_suite do
21:    for each t in tc.tests do
22:        ADDGENERATEDASSERTION(t)
23:        if PASSINGTESTS(r, tc) == False then
24:            REMOVEGENERATEDASSERTION(t)
25:        end if
26:    end for
27: end for
28: score_gen ← RUNPIT(r, green_suite)
29: for each tc in green_suite do
30:    for each t in tc.tests do
31:        ADDORIGINALASSERTION(t)
32:        if PASSINGTESTS(r, tc) == False then
33:            REMOVEORIGINALASSERTION(t)
34:        end if
35:    end for
36: end for
37: score_orig_gen ← RUNPIT(r, green_suite)
38: return ⟨ score_orig, score_without, score_gen, score_orig_gen ⟩
```

Chapter 5

Tracto: a Neuro-Symbolic Approach for Generating Concrete Oracles

In this chapter, we address the problem of automatically deriving concrete test oracles and we present TRACTO, a neuro-symbolic approach to derive concrete test oracles from commonly available software artifacts: source code and documentation. TRACTO generates test oracles token-by-token through an iterative process involving a symbolic module, which enforces syntactic and semantic constraints to increase compilability, and a neural module based on *Qwen2.5-Coder 3B* and fine-tuned with LoRA, which guides the generation toward valid assertions. The evaluation compares TRACTO against two baselines: a pure neural model fine-tuned to generate oracles in a single step without symbolic constraints, effectively isolating the contribution of the symbolic component, and the best-performing model from Chapter 4 as a vanilla baseline.

Chapter 3 indicates that LLM-generated axiomatic oracles capture invariant properties with some limited fault detection capability. Chapter 4 presents a comprehensive empirical study evaluating how vanilla large language models perform on concrete oracle generation. The mutation analysis indicates that the mutation score of LLM-generated concrete oracles (43%) is comparable to the mutation score of developer-written oracles (45%), showing the effectiveness of LLM-generated concrete oracles for fault detection. These results motivate the study of the automatic generation of concrete oracles.

This chapter investigates whether integrating symbolic constraints with fine-tuned neural generation can overcome the compilation and correctness limitations of purely neural approaches, building on the findings from Chapter 4. TRACTO (**TRA**nsformer-based token-by-token **C**oncrete **T**est **O**racle **G**eneration) generates test oracles token-by-token through an iterative interaction between a symbolic and a neural component. The symbolic module generates candidate tokens guided by grammar rules and project-specific context, enforcing validation constraints that prune incorrect oracles early to prevent invalid assertions.

The chapter defines TRACTO, and compares it against a pure-neural model fine-tuned to infer oracles in one step (thus isolating the symbolic component) and the vanilla baseline from Chapter 4 (*Qwen2.5-Coder 32B*), using a post-cutoff, unbiased test set of 3448 datapoints.

We trained TRACTO and the pure-neural model on the same oracles, by using the same

backbone (*Qwen2.5-Coder 3B*) and LoRA fine-tuning strategy [36]. The key difference lies in the learning granularity: the neuro-symbolic variant decomposes each oracle into token-level sequences to enable interaction with the symbolic component during the iterative generation, whereas the neural model learns how to reproduce complete oracles in a single step. This setup isolates the specific contribution of the symbolic component within the overall generation process. Although the pure-neural model achieves higher raw accuracy (31% exact match), TRACTO demonstrates the added value of symbolic integration in improving the oracle correctness. When the symbolic module successfully completes oracle generation without early termination, TRACTO achieves an 80% compilation rate and 59% test pass rate, outperforming the pure neural model (72% compilation, 51% test pass) and substantially exceeding the vanilla baseline (10% compilation, 7% test pass). However, the symbolic module discards approximately 43% of generation attempts (680 out of 1591 total attempts) upon detecting patterns that indicate divergence toward invalid oracles. These patterns include malformed literals, recurrent token sequences, and assertions with mismatched parameter counts. This filtering behavior reveals a fundamental trade-off: symbolic constraints improve the correctness of successfully generated oracles at the cost of reducing coverage, i.e., the total number of oracles produced.

The remainder of this chapter proceeds as follows.

Section 5.1 presents the architecture and the workflow of TRACTO, the neuro-symbolic approach to generate concrete oracles.

Section 5.2 details the symbolic module, including the ANTLR4-based grammar analysis for candidate token retrieval, the JavaParser integration for project-specific identifier resolution, and the validation constraints that proactively filter problematic patterns such as parameter count violations, recurrent sequences, malformed literals, and grammar inconsistencies.

Section 5.3 describes the neural module, including the fill-in-the-middle prompting strategy for token-by-token generation, the two-phase approach for literal generation, the LoRA-based parameter-efficient fine-tuning setup, and the decoding constraints applied during training and inference.

Section 5.4 outlines the procedure for building a comprehensive dataset from 135 Java projects, including oracle extraction through Algorithms 1 and 2 from Chapter 4, token-level decomposition for TRACTO's training set, the dataset splitting strategy, the prompt design, and the experimental setup comparing TRACTO against pure neural and vanilla baselines.

Section 5.5 reports the results of the experimental studies conducted to compare the accuracy of TRACTO against a pure neural baseline and the best vanilla model from Chapter 4 in terms of exact match and inclusion metrics; and (ii) evaluate the correctness of generated oracles through compilation rates and test pass rates, revealing TRACTO's superior per-oracle quality (80% compilation, 59% test pass) despite reduced coverage due to symbolic validation filtering.

Section 5.6 discusses the internal and external threats to validity that may affect the credibility and generalizability of the results, and outlines the measures adopted during the study to mitigate these risks.

5.1 Tracto Architecture

TRACTO generates concrete test oracles through an iterative token-by-token process that integrates a symbolic module for candidate retrieval and validation with a neural module for

token selection and literal generation. Figure 5.1 illustrates the overall workflow of TRACTO. TRACTO uses the symbolic module to enable the neural component to generate concrete literal values essential for assertions on specific test inputs, differently from TRATTO, which generates axiomatic oracles by restricting literal generation to enforce compilability. This architectural decision addresses the fundamental challenge of generating concrete oracles: inferring exact values (such as string, boolean, integer values) that match expected program outputs.

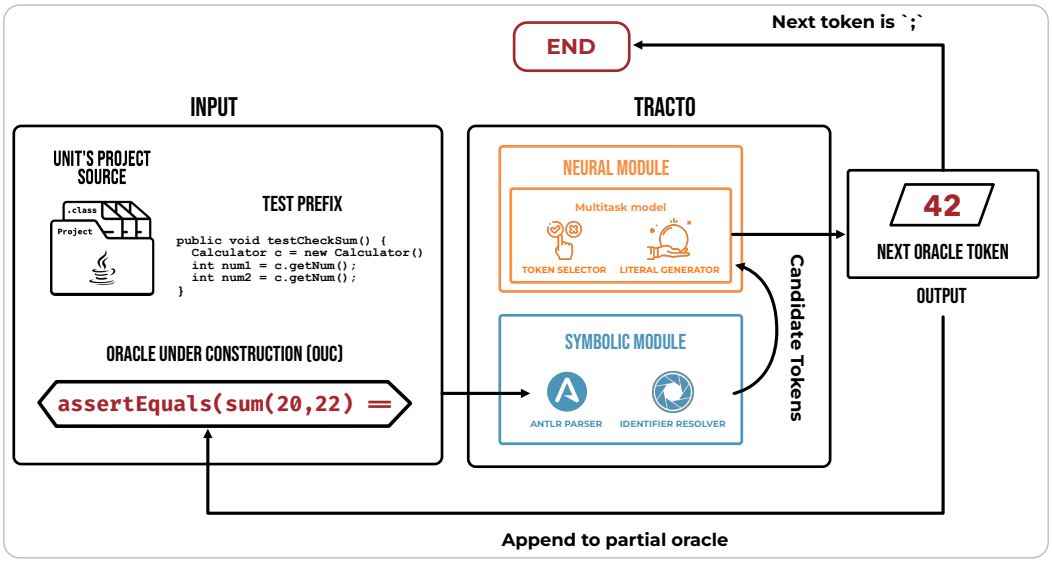

Figure 5.1. Workflow of Tracto.

TRACTO receives the source code of the project, the test prefix, the classpaths of the focal and the test classes, and the partial oracle (initially empty) as input.

The symbolic module analyzes the current generation state and retrieves valid candidate tokens for the neural module's selection process.

5.1.1 Oracle Generation Workflow

The oracle generation workflow proceeds iteratively until reaching completion or triggering early termination via validation constraints:

1. **Grammar Analysis**: The ANTLR Parser analyzes the Oracle Under Construction (OUC) using Java grammar rules to determine which token categories are syntactically valid and can follow the current position. ANTLR returns three types of tokens: (1) *Java tokens* comprising keywords, operators, symbols, and parentheses (2) *Generic literals* represented by abstract labels (such as `STRING_LITERAL`, `DECIMAL_LITERAL`, `BOOLEAN_LITERAL`, etc.); (3) The generic label `IDENTIFIER` when the grammar permits class names, method names, variable names, or attribute references specifically related to the project or its dependencies.

2. **Identifier Resolution**: When ANTLR returns the label `IDENTIFIER` among the list of eligible token types, the symbolic module activates the *Identifier Resolver* component. The

resolver processes the current state of the Oracle Under Construction (OUC) to retrieve all possible project-specific identifiers that can follow the partial oracle. For example, given a partial oracle containing an instance variable followed by a dot operator (`instance.`), the resolver retrieves concrete identifiers of methods and attributes defined within the instance's class, as these members can be accessed from the partial oracle. The *Identifier Resolver* attempts to resolve candidate tokens referring to classes, methods, and attributes from external libraries when dependency information is available. If the resolver fails to retrieve candidate tokens (typically due to unresolvable dependencies or missing classpath entries) only the Java tokens and literal labels pass to the neural component.

3. **Neural Token Selection**: The neural module operates as a multitask model that learns to predict the next token. In the first phase, the neural module receives the test prefix, the partial oracle, the three candidate token lists, and contextual information formatted as a fill-in-the-middle prompt. The model selects one token from the candidate lists and outputs it in JSON format: `{"choice": "<selected_token>"}`. During this phase, decoding constraints force the model to generate only valid tokens present in the candidate lists, preventing the selection of tokens that ANTLR and the *Identifier Resolver* did not identify as grammatically and semantically valid.

4. **Literal Generation**: The model produces the selected token unless it represents a generic literal (for instance `STRING_LITERAL`), in which case it advances to the second phase. The neural module receives a modified prompt that requests the generation of the concrete value for the selected literal type. The model generates a free response without decoding constraints, and produces the specific literal value to append to the partial oracles.

5. **Validation and Accumulation**: The symbolic module validates the generated token according to multiple criteria: parameter count consistency with assertion signatures, absence of illegal recurrent token patterns, literal syntax correctness, and grammar consistency. If the validation succeeds, TRACTO appends the generated token to the partial oracles, otherwise the symbolic module terminates the generation and discards the oracle under construction, preventing the inference of non-compilable or semantically invalid assertions.

6. **Termination**: The workflow terminates either successfully, when the neural module selects a semicolon (`;`) that indicates the successful completion of the assertion statement, or failing, when the symbolic validation constraints detect problematic patterns and trigger early termination to prevent divergence toward invalid oracles.

This validation integrated in the iterative process provides quality control mechanisms that purely neural approaches lack. The symbolic module's proactive filtering ensures that successfully generated oracles meet higher standards of syntactic correctness and semantic validity.

5.1.2 Architecture Limitations

The architecture imposes an important limitation: when the symbolic module does not include the target token in the candidate lists, the decoding constraints force the model to select a suboptimal alternative from available candidates. This limitation arises in two scenarios:

Incomplete Identifier Resolution: The Identifier Resolver fails to retrieve some valid identifiers due to unresolvable dependencies, missing classpath entries, or complex generic type

parameters. In these cases, the list of project related candidate tokens does not include the target token, and the model is forced to choose among available candidates due to the applied decoding constraints. This forced selection cannot prevent consequent compilation failures or semantic incorrectness.

ANTLR Grammar Coverage: ANTLR's Java grammar, while comprehensive, defines the syntactic rules that constitute valid Java code structures, but does not enforce semantic constraints. It specifies how tokens may be combined into grammatically correct statements (such as describing a method call as a sequence of a method name, an opening parenthesis, a list of comma-separated arguments, and a closing parenthesis) but it cannot verify whether the invoked method exists, nor whether the number and types of arguments match its signature. Consequently, the approach may still generate syntactically valid oracles that complies with grammar patterns but contains semantic inconsistencies.

Our dataset analysis reveals that the target token does not appear in the candidate lists retrieved by the symbolic module in 3% of datapoints. This incomplete coverage is a systematic gap that can constrain the neural module's ability to generate correct oracles. When incomplete coverage occurs during inference, the decoding constraints prevent the model from selecting the correct token, forcing suboptimal choices that can lead to wrong oracle generation.

5.2 Symbolic Module

The symbolic module implements three functionalities: it retrieves valid candidate tokens through grammar analysis, resolves identifiers to collect project-specific tokens, and applies validation constraints to ensure that generated tokens respect syntactic and semantic rules, to prevent deviations that would lead to incorrect oracle generation.

5.2.1 ANTLR Parser

TRACTO employs ANTLR4 (ANother Tool for Language Recognition) [56], a parser generator with support for complex grammars, automatic parse tree generation, and listener/visitor mechanisms for parse tree traversal. The ANTLR Parser component processes the partial oracle at each generation step to determine which token categories can validly follow according to Java syntax rules.

The parser operates with a comprehensive Java grammar that specifies valid token sequences, expression structures, statement patterns, and syntactic constructions. When parsing the partial oracle, ANTLR identifies the current syntactic context (whether the oracle occurs in a feasible position: a method call, a parameter list, or other structural element) and determines the categories of tokens that can be selected next without violating the grammatical correctness.

ANTLR returns (i) **concrete Java tokens** that constitute fixed elements of the Java programming language (such as keywords, operators, symbols, and parentheses) and returns their exact string representations, (ii) **generic literal labels** that are abstract type labels indicating literal categories at the current position, like `DECIMAL_LITERAL`, `BOOL_LITERAL`, `STRING_LITERAL`, and (iii) **generic identifier label**, the abstract label `IDENTIFIER` when the grammar permits class names, method names, variable names, or attribute references. ANTLR cannot determine concrete literal values: the generation falls back to neural component, required to generate the concrete value based on semantic understanding of the program's expected behavior. ANTLR is a parser and it determines that an identifier can appear at the current position but cannot spec-

ify which identifiers are valid (this operation requires semantic analysis of the project context, types, scopes, and accessibility rules that ANTLR's syntactic analysis cannot provide)

TRACTO introduces the label `LAMBDA_PARAM` when analyzing contexts where lambda expressions appear. TRACTO identifies Lambda parameters with names that it generates on the fly, and are not predetermined by the code structure, so the *Identifier Resolver* cannot statically retrieve them from the project context. The neural module must generate appropriate parameter names that maintain consistency within the lambda body.

The ANTLR oracle generation begins with an empty partial oracle and a list of all the valid assertion method names that TRACTO supports from JUnit, like '`assertTrue`', '`assertEquals`', '`assertNotNull`', and progresses by continuously update valid token categories based on the evolving syntactic context.

5.2.2 Identifier Resolver

When ANTLR returns the `IDENTIFIER` label among eligible token types, the *Identifier Resolver* retrieves concrete identifier values from the project context, Java libraries that support abstract syntax tree generation and symbol resolution for Java projects, with JavaParser and SymbolSolver [64]. These libraries analyze multiple information sources to determine which identifiers can validly follow the partial oracle:

Test Prefix Analysis: The resolver parses the test prefix to identify variable declarations, method parameters, field references, and local variable assignments. For each identifier, the resolver determines its type by analyzing the initialization expressions, the method return types, and the explicit type annotations. The resolver maintains a symbol table mapping identifier names to their types throughout the test prefix execution flow.

Partial Oracle Context: The resolver analyzes the Oracle Under Construction to determine the current syntactic context. For expressions of the form `<identifier>.`, the resolver determines that the next token must be a member (method or attribute) of the `<identifier>` type. The resolver retrieves the class definition corresponding to the `<identifier>` type and extracts all accessible members. For expressions following method calls (`object.method().`), the resolver determines the return type and retrieves members of the returned type.

Scope and Accessibility: The resolver filters retrieved identifiers based on scope and accessibility rules. It excludes private members of classes inaccessible from the test context. It includes package-private members only when the test resides in the same package as the defining class, and protected members when the test class inherits from the defining class or resides in the same package.

External Dependencies: The resolver attempts to resolve identifiers from external libraries and dependencies when classpath information is available. This resolution enables the neural module to reference API methods and classes from the libraries included in the project as dependencies. The resolution can fail due to missing JARs, incomplete dependency specifications, or complex generic type parameters that SymbolSolver cannot resolve. When resolution fails, the resolver excludes external identifiers from the *Project Tokens* list.

The *Identifier Resolver* returns a list of concrete identifier strings that the neural module can select during the token generation. These identifiers populate the *Project Tokens* category in the candidate lists presented to the neural module. When resolution succeeds, the *Project Tokens* list provides a comprehensive coverage of the valid identifiers, enabling the neural module to select appropriate class names, method names, variable names, and attributes that compile correctly and semantically align with the test's intent.

5.2.3 Validation Constraints

The validation constraints feature of the symbolic module monitors the generation process for patterns indicating divergence toward invalid oracles, beyond candidate token retrieval. The validation constraints step operates after each token generation step, analyzing the updated partial oracle to detect potential errors before they propagate through subsequent generation steps:

Parameter Count Validation: The parameter count feature step verifies that the number of parameters matches the assertion signature, when the neural module generates assertion method calls. The mechanism maintains a registry of assertion method signatures extracted from JUnit documentation. When ANTLR identifies an assertion method call in the partial oracle, the mechanism retrieves the expected parameter count and monitors subsequent token generation. After the neural module generates the final parameter, the constraint checks whether ANTLR indicates that additional parameters remain grammatically valid (signaled by a comma token). If the assertion method does not accept additional variable arguments and the neural module generates a further one, the constraint triggers termination.

Recurrent Pattern Detection: The symbolic module detects repetitive token sequences that indicate the neural module has incurred in an invalid generation loop. Common patterns include repeated opening parentheses without corresponding closures (`((((`), redundant operator sequences (`+ + + +`), cyclical method call chains (`object.method().object.method()`), , and repeated variable references without progression (`value.value.value`).

The detection mechanism maintains a sliding window over the most recent k tokens (with k sets to 10 in our experimental validation) and compares each new token against this window. If the new token creates a repetition of length $n > 3$ (the same n-token sequence appears three times consecutively), the constraint flags a recurrent pattern. The mechanism applies heuristics to distinguish genuine code patterns (chained method calls like `builder.add().add()` which are valid) from problematic repetitions (like `((((` which indicate parse errors).

5.3 Neural Module

The neural module implements two primary functions: it selects the next token from the set of candidates from the symbolic module, and generates concrete literal values when required. This section details the model architecture, fine-tuning approach, prompting strategy, two-phase generation workflow for literals, and decoding constraints.

5.3.1 Model Architecture and Fine-Tuning

TRACTO fine-tunes *Qwen2.5-Coder (3B)*, a code-specialized language model substantially smaller than the 32-billion-parameter model identified as optimal in Chapter 4. The decision to employ a smaller model stems from hardware constraints (the 3-billion-parameter variant represents the largest model that fits the available training infrastructure, that is a NVIDIA A100 GPU with 80GB VRAM) while maintaining reasonable training time. Research demonstrates that smaller models fine-tuned on task-specific data frequently achieve performance comparable to or exceeding that of larger general-purpose models, particularly when training data exhibits high

quality and task relevance[28, 71].

The fine-tuning process employs LoRA (Low-Rank Adaptation) [36] for parameter-efficient fine-tuning, which introduces trainable low-rank matrices into the attention layers of the model, while freezing the pre-trained weights. This approach dramatically reduces the number of trainable parameters, from billions to millions, enabling fine-tuning on the available hardware infrastructure. LoRA provides several advantages: parameter efficiency by decomposing weight updates into low-rank matrices, prevention of catastrophic forgetting by preserving pre-trained weights, computational efficiency by reducing GPU memory requirements and training time, and comparable performance to full fine-tuning across diverse tasks.

The training procedure fine-tunes the model for at least one epoch and up to three epochs, implementing an early stopping strategy based on validation set performance.

The training process computes loss only on the output tokens (the selected token in the `<|fim_middle|>` section), masking the input prompt from loss calculation (see 5.3.2). This approach focuses the model's learning on token selection and literal generation rather than prompt reconstruction.

5.3.2 Fill-in-the-Middle Prompting

The fill-in-the-middle prompting format structures the input to the neural module, and presents the oracle generation task as code completion conditioned on both preceding and succeeding context. Research demonstrates that fill-in-the-middle strategies optimize code completion performance by enabling models to leverage bidirectional context, and this format aligns with the pre-training regime of *Qwen2.5-Coder* [5].

The prompt structure consists of three sections delineated by special tokens: `<|fim_prefix|>` contains the test prefix and the oracle-under-construction generated so far, `<|fim_suffix|>` remains empty (as oracle generation extends the test rather than inserting text within existing statements), and `<|fim_middle|>` contains the output of the model.

Listing 5.1 (Phase 1) shows the prompt passed to the model for the token selection:

```
<|fim_prefix|>
// Test prefix
[test prefix code]
[Oracle Under Construction]<|fim_suffix|>

// The program uses JUnit Version: [version]

# NEXT TOKEN CANDIDATES
# PROJECT TOKENS: [list of identifiers]
# JAVA TOKENS: [list of keywords/operators/symbols]
# GENERIC TOKENS: [list of literal types]

# TASK: Choose exactly one among the three given lists
# as the NEXT token/type.
# OUTPUT: Generate a JSON on one line: {"choice":"<candidate>"}

# ADDITIONAL CONTEXTUAL INFORMATION (CAN GUIDE THE CHOICE):
// Invoked methods in test prefix
[javadoc, signature, and code]

// Methods defined in test class
[javadoc, signature, and code]
```

```

// Fields defined in test class
[field declarations]

// Methods defined in focal class
[javadoc, signature, and code]

// Fields defined in focal class
[field declarations]

<|fim_middle|>
```

Listing 5.1. Prompt template for phase 1.

Listing 5.2 (Phase 2) shows how the prefix section modifies the task description and removes the candidate lists to prompt the model to generate a concrete literal:

```
<|fim_prefix|>
// Test prefix
[test prefix code]
[Oracle Under Construction]<|fim_suffix|>

// The program uses JUnit Version: [version]

# EMIT a Java [LITERAL_TYPE] as the next token.
# OUTPUT: Generate a JSON on one line: {"choice":"<literal>"}

# ADDITIONAL CONTEXTUAL INFORMATION (CAN HELP THE GENERATION):
[same contextual information]

<|fim_middle|>
```

Listing 5.2. Prompt template for phase 2.

The contextual information corresponds to the P-C-T-F configuration identified in Chapter 4 as providing optimal oracle generation performance: test prefix (P), methods invoked in test prefix (C), test class information (T), and focal class information (F). This configuration supplies sufficient semantic context for accurate token selection while avoiding diminishing returns from additional information.

The input context window allocates 8,192 transformer tokens, and the output context window allocates 1,024 transformer tokens. The substantial output allocation accommodates long string literals that may span hundreds of characters.

5.3.3 Two-Phase Literal Generation

TRACTO implements a specialized workflow for generating literals in two phases: *token selection* and *literal generation*. The token selection phase identifies the literal type. The literal generation phase generates the concrete value for the selected literal type, be queering the neural module with a modified prompt to produce a Java literal of the specified type. While the token selection phase constrains the model to select from predefined literal types, the literal generation phase operates without decoding constraints, allowing the model to generate any syntactically valid literal value. This unconstrained generation enables the model to produce complex literals including multi-line strings, integers, floating-point values, and hexadecimal constants.

The symbolic module validates the syntax of the literal values from the neural model according to Java grammar rules.

This two-phase approach addresses the fundamental difficulty of producing concrete values that precisely match the expected outputs. String literals exemplify this challenge: in assertion statements a generated string must match the target string character-for-character, including whitespace, punctuation, and special characters. Semantic equivalence without syntactic identity leads to assertion failures.

5.3.4 Decoding Constraints

The decoding constraints feature of TRACTO ensures that the model produces syntactically valid outputs, during the token selection and literal type selection steps of the token generation phase. The output vocabulary of the model restricts to only those tokens present in the candidate lists provided by the symbolic module, when selecting concrete tokens or literal types. The constraint mechanism implements prefix-based filtering: as the model generates each character of the JSON output, the decoder eliminates tokens whose string representations cannot extend to valid candidates.

The constraint implementation operates on the model's output logits at each generation step. Before sampling or selecting the next token, the constraint mechanism identifies which vocabulary tokens could extend the current partial output to match one of the provided candidates. For vocabulary tokens that cannot extend to any valid candidate, the mechanism sets their logits to negative infinity, effectively removing them from the sampling distribution.

The constraint implementation maintains sequences of valid token sequences to guarantee correct JSON formatting throughout generation. The output must form valid JSON objects with the structure `{"choice":"<candidate>"}`, where `<candidate>` matches one of the provided tokens exactly. The decoder ensures that opening braces match closing braces, quotation marks appear in pairs, and the colon separator appears between the key and value.

The model generates concrete values without decoding constraints, in the second phase of literal generation. The prompt instructs the model to produce a literal of the specified type, and the model produces the value based on contextual understanding of the program semantics. Unconstrained generation allows the model to produce any valid literal, including complex strings with arbitrary length, escape sequences, unicode characters, and formatting conventions. This flexibility proves essential for generating literals that match expected program outputs, but introduces the risk of syntactic errors that the symbolic validation constraints subsequently detect.

5.4 Dataset, Prompt, and Experimental Setup

This section describes the comprehensive dataset generation process from 135 Java projects, including oracle extraction, token-level decomposition, dataset splitting, and encoding strategies. The section also details the pure neural baseline configuration and the evaluation setup.

5.4.1 Oracle Extraction from Java Projects

The dataset construction begins with the 135 Java projects that we identify in Chapter 4 through the repository selection process. These projects satisfy quality criteria including minimum commit counts (at least 100 commits), issue counts (at least 10 issues), contributor counts (at least

5 contributors), and star counts (at least 10 stars), and contain at least one commit between September 1, 2024, and May 5, 2025.

Algorithm 2 from Chapter 4 processes the commit history of each project to identify test cases added or modified after September 1, 2024, ensuring temporal separation from the training cutoff dates of the large language models evaluated in this study. The algorithm extracts the final version of each test case by tracking additions, deletions, and modifications across commits.

Algorithm 4 from Chapter 4 processes the extracted test cases to build a dataset of oracles. For each test case, the algorithm identifies all assertion statements with pattern matching and abstract syntax tree analysis, treats each assertion as a distinct oracle, and extracts the test prefix—the code preceding the assertion statement within the same test method. The algorithm also retrieves contextual information including: the javadoc, signature, and implementation of methods invoked within the test prefix; the javadoc, signature, and implementation of methods defined in the test class; and the javadoc, signature, and implementation of methods defined in the focal class.

The oracle extraction process yields a dataset of complete assertions represented as strings. Each oracle corresponds to one assertion statement from one test case, capturing the full syntactic structure from the assertion method name through the closing semicolon.

5.4.2 Token-Level Decomposition

TRACTO decomposes each complete oracle into a sequence of token-level training instances using ANTLR4 and JavaParser. The decomposition process generates training datapoints that capture the token-by-token generation trajectory, where each datapoint represents one step in constructing the oracle from an empty string to the complete assertion.

The decomposition algorithm begins by initializing the oracle under construction as an empty string. It then parses the partial oracle using ANTLR4 to determine valid token categories according to Java grammar rules. For tokens categorized as *IDENTIFIER*, the algorithm invokes JavaParser and SymbolSolver to retrieve concrete identifier values from the project context. Next, the algorithm constructs three candidate lists—*Java Tokens*, *Project Tokens*, and *Generic Tokens*—which represent the possible continuations for the current partial oracle. It then extracts the next token from the complete oracle, which serves as the target token. A training datapoint is created containing a tuple in the form of $\langle$*(tp, jt, pt, gt, ouc, t)*$\rangle$, where *tp* refers to the test prefix, *jt,pt, and gt* are the three lists of java, project and generic tokens, *ouc* reports the oracle under construction, and *t* is the target oracle.

After generating the datapoint, the algorithm appends the target token to the oracle under construction and repeats the process from the parsing step until the complete oracle has been fully decomposed.

For tokens categorized as generic types, the decomposition creates two sequential training datapoints. The first datapoint targets the literal type label (e.g., `STRING_LITERAL`), and the second datapoint targets the concrete literal value.

The decomposition process generates 662,485 token-level datapoints from the oracle dataset. Analysis of target token coverage reveals that in 96% the target token appears within the candidate lists retrieved by the symbolic module. In 4%), the target token does not appear in the candidate lists, revealing limitations in the symbolic module's token coverage.

Dataset Splitting Strategy

The dataset splitting strategy allocates the 135 projects to training, validation, and test sets while considering both project size and token datapoint counts. The splitting algorithm uses stratified sampling to maintain representative distributions. The resulting split assigns 87 projects (65%) to the training set, 14 projects (10%) to the validation set, and 35 projects (25%) to the test set.

Evaluation Dataset and Approaches

The test set includes 35 projects (25%). To ensure complete separation from training data, the evaluation restricts analysis to test cases added after September 1, 2024. Algorithm 1 identifies these post-cutoff test cases, and Algorithm 2 extracts the corresponding oracles, resulting in a test set of 3,448 oracles. The experiment involved 3 different approaches:

- **Pure Neural Model**: Qwen2.5-Coder (3B) fine-tuned to generate complete oracles in a single inference step.
- **TRACTO (Neuro-Symbolic)**: Token-by-token generation through iterative interaction between the symbolic module (ANTLR Parser and Identifier Resolver with validation constraints) and the neural module represented by Qwen2.5-Coder (3B) fine-tuned to generate the next target token.
- **Vanilla Qwen2.5-Coder (32B)**: The best-performing vanilla model from Chapter 4, queried with the P-C-T-F prompt configuration.

5.5 Experimental Evaluation

This section presents the results of a comprehensive experimental validation that addresses two research questions:

RQ1 ***Accuracy comparison****: How does the neuro-symbolic* TRACTO *approach compare with a fine-tuned pure neural model and a large vanilla LLM in terms of oracle generation accuracy on an unbiased test set?* We compare the neuro-symbolic TRACTO approach with purely neural and large vanilla LLM baselines to assess how symbolic integration influence the accuracy of oracle generation

RQ2 ***Correctness comparison****: How does the neuro-symbolic* TRACTO *approach compare with a fine-tuned pure neural model and a large vanilla LLM baselines in terms of oracle correctness?* We assess how the integration of symbolic module within TRACTO affects the correctness of generated oracles—measured through compilation rate and test pass rate, relative to purely neural and large vanilla LLM baselines.

5.5.1 RQ1: Accuracy Comparison

RQ1 evaluates the accuracy of three approaches in terms of exact match and inclusion metrics. The exact match accuracy measures the percentage of generated oracles that match the target oracle character-for-character. The inclusion accuracy measures the percentage of generated

oracles that contain the target oracle as a substring.

Results

Table 5.1 presents the accuracy results for the three approaches.

Table 5.1. Accuracy of the approaches.

Approach	Exact Match	Inclusion
Pure Neural (3B)	31.0%	33.6%
TRACTO (3B)	12.9%	14.1%
Vanilla (32B)	23.3%	26.1%

The pure neural model reaches 31% exact match accuracy and 33.6% inclusion accuracy, TRACTO achieves 12.9% exact match and 14.1% inclusion, the vanilla Qwen2.5-Coder (32B) achieves 23.3% exact match and 26.1% inclusion, on the dataset of 3,448 test cases.

Pure neural fine-tuning achieves highest raw accuracy: The accuracy of the pure neural model is 20 percentage points higher than the accuracy of TRACTO. This gap indicates that the oracle generation without symbolic constraints during inference produces outputs that more closely match developer-written oracles in terms of syntactic structure.

Fine-tuning provides substantial benefits: The exact match accuracy of the pure neural model is 6 percentage points higher than the accuracy of the vanilla baseline, indicating that task-specific fine-tuning enables smaller models to outperform larger general-purpose models.

TRACTO's accuracy-coverage trade-off: TRACTO's lower accuracy scores reflect the trade-off between generation coverage and oracle quality, as we will see from the results of RQ2. The symbolic validation constraints that improve correctness also reduce the total number of successfully generated oracles. When the symbolic module detects divergence patterns, it aborts the generation, resulting in no oracle for that test case.

Answer to RQ-1:
The pure neural model achieves the highest raw accuracy (31.0% exact match, 33.6% inclusion), outperforming TRACTO (12.9%, 14.1%) and the vanilla baseline (23.3%, 26.1%). The raw accuracy measures the syntactic similarity without accounting for quality differences. The lower accuracy of TRACTO reflects its proactive filtering strategy: the symbolic module discards generation attempts when detecting signs of divergence, reducing coverage, while improving the correctness of successfully generated oracles, as shown in RQ2.

5.5.2 RQ2: Correctness Evaluation

RQ2 evaluates the practical correctness of generated oracles by measuring the success of the compilation and the outcomes of the test execution.

Experimental Procedure

We evaluate the correctness for each project in the test set as follows. We create a *baseline version* by removing the developer-written assertion statements from the original test cases. This baseline preserves the original test logic allowing the evaluation of the correctness to focus on the effect of the oracles generated by the three target approaches, when integrated within the test prefixes. The evaluation then proceeds through the following steps:

1. We compile the original baseline test cases without oracles to verify baseline compilability and execution.

2. For each approach, we replace the target oracle with the generated oracle, we compile the modified test file, and if compilation succeeds, we execute the test and record pass/fail outcome.

We exclude from analysis the projects that fail to compile, single test cases whose baseline version fails to compile, and test cases whose baseline version fails execution.

Successfully compiling 16 out of 35 test set projects, the correctness evaluation analyzes the subset of test cases from these projects that satisfy the filtering criteria.

Results

Table 5.2 shows the compilation and test pass rates of the three approaches computed on 16 projects. The rates for TRACTO indicate the percentage of compiling and passing test cases relative to the successfully completed generations, excluding the oracles that do no pass the symbolic validation (680 out of 1591 total attempts, 43%).

Table 5.2. Average compilation and test pass rates on 16 projects.

Approach	Total Oracles	Compilation Rate	Test Pass Rate
Pure Neural (3B)	1,591	72.6%	51.3%
TRACTO (3B)	911	80.4%	59.1%
Vanilla (32B)	1,538	10.5%	7.2%

Table 5.3 shows the results per-project for the 16 successfully compiled projects. The table indicates the number of generated oracles (*G*), the number and percentage of oracles that successfully compile (*C*) and the number and percentage of the test case that pass when executed (*TP*).

The pure neural model generates 1,591 oracles with 73% compilation rate and 51% test pass rate. TRACTO completes 911 generations (after discarding 835 attempts) with 80% compilation rate and 59% test pass rate among completed oracles. The vanilla baseline generates 1,538 oracles with 10% compilation rate and 7% test pass rate.

TRACTO achieves the highest compilation and test pass rates with respect to the successfully completed generations, TRACTO outperforms the pure neural model both in compilation rate (80% versus 73%) and in test pass rate (59% versus 51%). This indicates that symbolic validation constraints successfully identify and filter problematic generation paths.

Symbolic validation trades coverage for quality: TRACTO's symbolic module aborts 43% of generation attempts (680 out of 1591) when detecting signs of divergence, reflecting in a low false positive rate and a lower total generation coverage than the the pure neural model.

Both fine-tuned approaches dramatically improve over vanilla baseline: both TRACTO and the pure neural model substantially outperform the vanilla 32B baseline. TRACTO achieves 69.9 percentage points higher compilation rate and 51.9 percentage points higher test pass rate compared to the vanilla baseline.

Coverage–correctness trade-off: The comparison reveals a Pareto frontier: TRACTO produces fewer oracles but with higher correctness (80% compilation, 59% test pass, 911 oracles) that the pure neural model, which generates more oracles with lower correctness (73% compilation, 51% test pass, 1,591 oracles) than TRACTO.

Answer to RQ-2: TRACTO achieves the best compilation and pass rates: 80% compilation and 59% test pass rate, outperforming the pure neural model (73%, 51%) and substantially exceeding the vanilla baseline (10%, 7%). The good compilation and pass rates comes with a reduced generation coverage: TRACTO infers 911 oracles, less than the pure fine-tuned neural model (1,591), as the symbolic module discards 680 attempts (43%) when detecting invalid patterns. The results show that symbolic validation provides measurable value by proactively filtering diverging generation paths, improving the reliability of the successfully generated oracles, while indicating a fundamental trade-off between test completion coverage and quality.

	Tracto					Qwen2.5-Coder (32B)					Qwen2.5-Coder (3B-finetuned)				
	G	C	%	TP	%	G	C	%	TP	%	G	C	%	TP	%
jenkinsci/ansicolor-plugin	94	71	76	68	72	187	17	9	17	9	186	176	95	86	46
jenkinsci/lockable-resources-plugin	63	63	100	46	82	89	5	6	4	5	101	90	89	74	79
datadog/java-dogstatsd-client	5	4	80	0	0	5	0	0	0	0	6	5	83	0	0
crawler-commons/crawler-commons	35	31	89	22	67	35	5	14	5	15	27	25	93	24	96
42bv/beanmapper	43	15	35	6	18	54	8	15	7	16	55	21	38	16	35
cyclonedx/cyclonedx-core-java	224	186	83	49	22	277	17	6	14	5	278	134	48	67	24
networknt/json-schema-validator	290	232	83	37	0	566	68	12	0	0	567	402	72	114	50
ibm-messaging/kafka-connect-mq-source	15	6	40	2	13	14	2	14	2	14	15	14	93	13	87
netarchivesuite/solrwayback	3	3	100	1	33	2	1	50	1	50	3	3	100	1	33
getodk/javarosa	12	8	67	7	58	22	0	0	0	0	23	10	43	9	39
seleniumhq/htmlunit-driver	44	34	77	30	68	45	1	2	0	0	46	43	93	39	85
jenkinsci/azure-storage-plugin	56	51	91	40	71	81	13	16	13	16	82	59	72	51	62
osiris-team/autoplug-client	3	2	67	2	67	7	1	14	1	14	8	7	88	5	63
jenkins-infra/repository-permissions-updater	5	4	80	4	80	4	0	0	0	0	4	1	25	1	25
photon-github/anticheataddition	19	13	68	5	28	7	3	43	3	50	42	37	88	27	66
datastax/cassandra-data-migrator	0	0	0	0	0	143	19	13	17	13	148	120	81	99	71
Total	911	723	**80**	319	**59**	1538	160	**10**	84	**7**	1591	1147	73	626	51

Table 5.3. Number of oracles generated (G), that compile (C) and test case that pass (TP)

5.6 Threats to Validity

TRACTO represents a preliminary exploration of neuro-symbolic approaches for concrete oracle generation. The work is still unpublished and has not undergone the rigorous peer review process, which limits the maturity of our findings and constrains the generalizability of our results. This section discusses both internal and external threats to validity that may affect the credibility and generalizability of the evaluation, along with the measures adopted to mitigate these risks.

Internal Validity

The internal validity of the TRACTO evaluation is challenged by complexities in model selection, the fidelity of the symbolic instrumentation, and the mechanisms of data preparation.

The experimental design compares TRACTO against two primary baselines: a fine-tuned pure neural model sharing the same 3-billion-parameter backbone, and a 32-billion-parameter vanilla model (**Qwen2.5-Coder**). While this setup aims to isolate the contribution of the symbolic component, several confounding variables introduce ambiguity into the causal interpretation of the results. The decision to utilize the *Qwen2.5-Coder 3B* model as the neural backbone for TRACTO was driven by hardware constraints, specifically the necessity to fit the training

and inference processes within the VRAM limits of a single NVIDIA A100 GPU. This resource constraint required a comparison against a vanilla baseline (32B) that is an order of magnitude larger. Literature on large language models consistently demonstrates that reasoning capabilities, instruction following, and the ability to handle complex contexts scale non-linearly with parameter count. The vanilla 32B model possesses inherently superior "world knowledge" and coding intuition compared to the 3B backbone used in TRACTO. The superior compilation rate of TRACTO (80%) versus the vanilla 32B model (10%) is a strong signal of the value of fine-tuning and the contribution of the symbolic module. The comparison with a pure neural model, sharing the same 3B configuration isolate the contribution of the symbolic module of the neuro-symbolic approach, improving the compilation rate by 7%. However, the comparison is still asymmetric. It remains unclear whether a neuro-symbolic 32B model would exhibit linear gains or diminishing returns. If the symbolic module acts primarily as a guardrail for "weaker" models, its utility might decrease as the underlying neural backbone becomes more capable of self-correction. Conversely, if the symbolic module provides information inaccessible to the neural weights (e.g., precise project identifiers), the gains should persist. The current experimental design cannot disentangle these effects, leaving the scalability of the neuro-symbolic benefit as an open internal validity threat.

Static analysis in Java is notoriously difficult due to dynamic features such as reflection, complex generics, and dependency injection frameworks. TRACTO's *Identifier Resolver* attempts to construct a comprehensive list of valid tokens (methods, fields, classes) available at a given cursor position. The dataset analysis revealed that in 4% of the training datapoints, the actual target token from the developer-written oracle was missing from the candidate lists generated by the symbolic module. This implies that in 1 out of every 25 generation steps, the symbolic module presented the neural module with an unsolvable problem: *choosing the correct token from this list*, when the correct token was not in the list. This forced error phenomenon creates a ceiling on TRACTO's potential accuracy that is unrelated to the neural model's capability. In the pure neural baseline, the model can "hallucinate" the correct token even if it resides in a complex dependency that static analysis missed. In TRACTO, the decoding constraints actively prevent the model from selecting that correct token if the symbolic module missed it. This instrumentation threat artificially deflates the accuracy of TRACTO compared to the pure neural baseline, confounding the assessment of the neuro-symbolic architecture's true efficacy.

A defining feature of TRACTO is its validation mechanism: if the symbolic module detects a diverging pattern (e.g., a method call with incorrect arity), it aborts the generation process. This mechanism resulted in the abortion of 680 out of 1591 generation attempts (approximately 43%). Comparing the compilation rate of TRACTO (calculated on $N = 911$) against the pure neural model (calculated on $N = 1591$) introduces a selection bias. The pure neural model is penalized for attempting difficult cases that TRACTO skips. While this reflects the practical utility of the tool (it is better to produce nothing than garbage), it complicates the theoretical comparison.

Finally, the manual validation of the 3448 generated oracles is almost impossible. We automatically validated the generated oracles by comparing them with the available and carefully designed oracles. We mitigated the risks of incorrect validation by implementing the comparison with string matching and inclusion that misses valid oracles that differ from the reference manual oracles only for even small syntactic elements. We decided to report conservative results that are a pessimistic under-approximation of the actual data, to avoid any risk of overestimating the results.

External Validity

The evaluation focuses on projects retrieved from GitHub with specific quality filters: at least 100 commits, 50 issues, 10 distinct contributors, 10 stars, and Maven-based compilation. This selection process creates a bias toward mature, well-maintained open-source projects with established testing practices. The findings may not generalize to smaller projects with less formal development processes, proprietary codebases, or projects using non-standard build systems. Additionally, the evaluation excludes projects with external build dependencies or complex transitive dependency chains, potentially limiting applicability to real-world scenarios where build infrastructure varies substantially.

TRACTO's symbolic module imposes computational overhead due to iterative grammar-based validation at each token generation step. The approach requires multiple parsing passes through the partial oracle and candidate token filtering operations, resulting in longer oracle generation times compared to single-step neural generation. While the neural module and symbolic module are decoupled via a REST API, facilitating potential deployment on separate computational resources, this modularity introduces network latency that may impede practical adoption in resource-constrained or real-time testing scenarios. The trade-off between symbolic correctness and computational efficiency remains a practical limitation for operational deployment.

The training data construction methodology relies on automated extraction of test cases created after September 1, 2024, to ensure temporal separation from LLM training cutoffs. However, this temporal cutoff assumes that publicly declared training cutoff dates are accurate and that no data leakage occurred through non-standard channels such as API queries, web scraping, or inclusion in fine-tuning datasets for commercial models. While we mitigate this risk by relying exclusively on open-source models with well-documented training procedures, the assumption of clean temporal separation may not hold universally.

Beyond temporal cutoff boundaries, a more fundamental threat to validity arises from the problem of structural data leakage: even test cases created after LLM training cutoffs may share syntactic patterns, assertion structures, and semantic properties with test cases present in the training data. Modern test case generation follows established conventions—such as the use of common assertion methods (e.g., assertEquals, assertTrue, assertNull), standard test naming patterns, and predictable assertion argument structures—that are likely represented in the training data of any LLM pre-trained on substantial code corpora. Consequently, even ostensibly novel test cases can implicitly encode patterns that the LLM has previously encountered, conferring an unfair advantage that is difficult to detect or measure. For instance, an assertion of the form `assertEquals(expected, actual)` with specific variable names and types may constitute a nearly identical pattern to assertions in the training data, even if the particular method under test is novel. This structural similarity cannot be easily remedied through temporal filtering alone, as it stems from the fundamental nature of standardized test writing practices across the software engineering community. The quantitative impact of this structural leakage on TRACTO's performance cannot be precisely determined, but we acknowledge that the superiority of neural models in oracle generation may be partially attributable to implicit memorization of common test patterns rather than genuine generalization to unseen oracle forms. This threat particularly affects the comparison between TRACTO and the pure neural baseline, as both approaches may benefit equally from such implicit pattern recognition, potentially masking differences in their respective capabilities on truly novel oracle patterns that deviate from conventional test writing practices.

The evaluation dataset is biased toward specific assertion types and oracle patterns that

predominate in mature Java projects. Rare assertion types, domain-specific assertion libraries, or testing frameworks diverging from standard JUnit conventions may be underrepresented in the evaluation, potentially inflating performance metrics on common assertion patterns while obscuring challenges on less frequent oracle forms.

Preliminary Nature and Maturity of the Work

The evaluation presented in this chapter represents a preliminary empirical investigation of neuro-symbolic approaches for concrete oracle generation. The work has not undergone peer review, and the methodology, experimental design, and findings remain subject to refinement and revision.

The mutation testing analysis in Chapter 4 establishes that LLM-generated concrete oracles can meaningfully contribute to mutation score improvements. However, TRACTO's integration into real-world testing workflows, its impact on developer productivity, and its practical utility for bug detection have not been empirically validated yet.

Together, these limitations underscore that this chapter presents preliminary findings from an ongoing research effort. Until such additional evidence is gathered, the conclusions drawn from this preliminary evaluation should be interpreted with caution.

Chapter 6

Conclusion

This dissertation defines a neuro-symbolic approach to automatically generate test oracles. The research systematically investigates whether integrating symbolic testing techniques with neural models produces semantically relevant test oracles with improved soundness and completeness compared to purely neural approaches.

Our systematic three-stage empirical investigation spanning both axiomatic and concrete oracle generation provides strong evidence supporting this hypothesis. The results show that symbolic constraints reduce false positives, improve compilation rates, and enhance test pass rates.

6.1 Contributions

This dissertation contributes to the state of the art in automated test oracle generation with TRATTO, a neuro-symbolic axiomatic oracle generator, a large-scale empirical study of LLMs for concrete oracle generation, and TRACTO, a neuro-symbolic concrete oracle generator.

6.1.1 Tratto: Neuro-Symbolic Axiomatic Oracle Generation

TRATTO, a neuro-symbolic approach to generate axiomatic test oracles, substantially improves axiomatic oracle generation. The approach reformulates the oracle problem as a token generation problem, where a symbolic module constrains the search space to ensure compilability while a neural module guides the generation toward semantically relevant oracles. TRATTO achieves 73% accuracy, 72% precision, and 61% F1-score on ground-truth datasets, outperforming both the symbolic baseline Jdoctor (61% accuracy, 62% precision, 25% F1-score) and the neural baseline GPT-4 (40% accuracy, 24% precision, 37% F1-score). The approach generates three times more correct oracles than Jdoctor while producing ten times fewer false positives than GPT-4, confirming that symbolic constraints effectively reduce spurious assertions without sacrificing the generalization capabilities that neural models provide.

The multitask learning framework of TRATTO couples oracle evaluation with token selection, enabling the model to decide whether an oracle can be generated and to choose tokens that form semantically coherent assertions. Our ablation study indicates that removing the symbolic module reduces accuracy by 6 percentage points, while eliminating the multitask architecture causes

a further 3-point drop. These results isolate the contribution of each architectural component and confirm that both symbolic constraints and multitask learning enhance oracle quality.

When integrated with EvoSuite-generated test suites, TRATTO improves mutation scores in five out of six projects for test suites containing implicit oracles, and in five projects for combined implicit and regression oracle suites. Jdoctor improves four test suites with implicit oracles but shows no gains with combined oracles. These results indicate that the broader applicability of TRATTO translates into measurable improvements in test suite effectiveness.

6.1.2 Large-Scale Empirical Study of LLMs for Concrete Oracle Generation

Our large-scale empirical study systematically evaluates vanilla large language models for concrete oracle generation, establishing baselines and identifying key factors affecting oracle quality. The study examines 13,866 test oracles from 135 Java projects, with all test cases created after model training cutoff dates to prevent data leakage. The experiments generate 554,640 predictions across multiple model families, sizes, and prompt configurations.

The results indicate that larger models consistently outperform smaller models across all specializations, while providing additional contextual information beyond test prefix and invoked method signatures yields no significant performance improvements. This finding challenges the common assumptions about optimal prompting strategies and suggests that model limitations derive more from fundamental architectural constraints than from insufficient input information.

Our mutation analysis shows that the fault-detection capability of LLM-generated oracles is comparable to developer-designed oracles (43% versus 45% mutation score) validating their practical utility for test suite augmentation. The study also reveals an important limitation: 50% of generated oracles in the experiment fail to compile or execute, highlighting substantial correctness issues that limit deployment in production testing environments.

The empirical study establishes methodological practices for rigorous LLM evaluation in code generation tasks. The emphasis on post-training-cutoff datasets addresses data leakage concerns prevalent in LLM research. The systematic variation of prompt configurations and model types provides a template for comparative evaluation that isolates the impact of individual experimental factors.

6.1.3 Tracto: Neuro-Symbolic Concrete Oracle Generation

TRACTO, a neuro-symbolic approach to generate concrete oracles, combines ANTLR4-based grammar analysis and JavaParser-based identifier resolution with neural token selection and literal inference. The systematic comparison against a fine-tuned pure neural model (*Qwen2.5-Coder-3B*) and the best-performing vanilla model from the empirical study (*Qwen2.5-Coder-32B*) on 3,448 post-cutoff test oracles reveals fundamental trade-offs between generation strategies.

The raw accuracy of the pure neural model (33%) is higher than the accuracy of both TRACTO (20%) and the vanilla LLM (10%). However, TRACTO yields more robust oracles tht both other approaches, with the highest compilation (80% versus 73% for the pure neural model and 10% for the vanilla LLM) and higher test-pass rates (59% versus 51% and 7%, respectively). These results support the research hypothesis that symbolic constraints measurably improve oracle correctness metrics (successful compilation and execution).

Since the neural component of TRACTO and the fine-tuned neural model share the same architecture and training data, their comparison effectively serves as an ablation study, to isolate the effect of symbolic constraints. The 7% compilation rate improvement and 8% test pass rate improvement directly attributable to symbolic integration confirm that neuro-symbolic approaches enhance the quality of oracles with respect to pure neural approaches.

The test oracle completion of TRACTO is lower than pure neural approaches: TRACTO aborted 43% of the generated oracles in the correctness evaluation experiment, since the symbolic validation discards the generation when detecting irrecoverable errors. This trade-off reflects a fundamental tension in neuro-symbolic design: stricter symbolic constraints improve per-oracle quality while reducing the proportion of test cases for which oracles are successfully generated.

6.2 Future Directions

The results presented in this thesis open new research directions towards dynamic contextual prompting with token-by-token refinement—an approach that fundamentally reconceptualizes how contextual information guides oracle synthesis.

6.2.1 Dynamic Input Prompts with Token-by-Token Context Refinement

Pure neural approaches require embedding all contextual information within a single static prompt because they generate the complete oracle in a single iteration. This constraint limits both the quantity and specificity of information available during generation. Our results show that large language models suffer from the "lost in the middle" phenomenon: when presented with extensive context windows, models struggle to effectively utilize the information located in the middle of long inputs, exhibiting a U-shaped attention bias that prioritizes information at the beginning and end while degrading performance on mid-context information [44]. Studies show that LLM performance degrades significantly as context length increases, with models dropping below 50% of their short-context performance when processing contexts of 32,000 tokens or more [34].

A token-by-token approach exploits the iterative generation process to update contextual information dynamically at each step, thus progressively refining context as the oracle takes shape. A token-by-token approach provides focused, targeted information relevant to the specific generation step, rather than overwhelming the model with extensive contextual information that degrades attention and introduces noise. By limiting context to immediately relevant information at each iteration, the approach maintains the focus of the model, and prevents the performance degradation associated with bloated context windows.

6.2.2 Enhanced Symbolic Constraints and Adaptive Neural Fallback

The symbolic component of a neuro-symbolic concrete test oracle approach can be strengthened through fine-grained semantic analysis that extends beyond basic grammar-based token retrieval and identifier resolution. Current implementations rely on grammar rules to enumerate candidate tokens and employ identifier resolvers to collect project-specific identifiers. However, this procedure overlooks semantic constraints derivable from static analysis that could filter out semantically invalid candidates before presenting them to the neural model.

For example, when the partial oracle invokes a method, the symbolic component can analyze parameter types and counts, eliminating candidates that cannot be passed as arguments to the method invocation. In this context, if a method expects an argument of type `String`, the symbolic analyzer can discard candidates referring to instances of incompatible types such as `Integer` or `List<Double>`. Similarly, when a method signature requires three parameters, the symbolic component can prevent the neural model from considering method calls with different arity. This fine-grained type analysis reduces the search space by filtering semantically invalid tokens before the neural model makes selection decisions, limiting the probability of incorrect generation. This contextual awareness transforms the symbolic component from a passive token enumerator into an active semantic validator that guides generation toward correct assertion structures.

Despite these improvements, the symbolic component will inevitably fail to resolve symbols when generating invocations to methods from external dependencies outside the analysis scope. Current neuro-symbolic approaches handle this limitation by providing a restricted set of fallback tokens, forcing the model to select from suboptimal candidates. An improved strategy treats such scenarios as opportunities for the neural model to operate independently, temporarily relaxing decoding constraints and allowing the model to predict tokens without symbolic restrictions.

This adaptive fallback mechanism requires training the neural model on dual objectives: token-by-token generation under symbolic constraints (when resolution succeeds) and complete oracle generation without constraints (when resolution fails). The dual training regime ensures the model develops competence in both constrained synthesis—where symbolic information guides each generation step—and unconstrained generation—where the model relies entirely on patterns learned during pre-training and fine-tuning. For popular external libraries such as JUnit, Mockito, or Apache Commons, the model likely encountered extensive usage examples during pre-training. When symbolic resolution fails for methods from these libraries, the model can infer correct tokens by leveraging its learned knowledge of common API usage patterns.

This integrated approach addresses both the symbolic component's limitations and the test oracle completion reduction derived from constrained generation. By combining enhanced semantic filtering with adaptive neural fallback, the system maintains high oracle quality when symbolic information is available while gracefully degrading to pure neural generation when symbolic analysis proves insufficient. The result is a neuro-symbolic architecture that prevents the symbolic component's limitations from unnecessarily constraining generation when the neural model possesses sufficient knowledge to proceed independently, while still leveraging symbolic precision whenever possible to improve oracle correctness.

6.3 Methodological Implications

This dissertation advances methodological practices for evaluating neural code generation systems. The emphasis on post-training-cutoff datasets addresses data leakage concerns that threaten the validity of LLM evaluations. The large-scale empirical study varies prompt configurations and model types in a systematic manner, creating a methodological design model that supports rigorous comparative evaluation. The ablation study design isolating symbolic components through matched neural baselines demonstrates how to empirically quantify the contribution of individual architectural elements in complex neuro-symbolic systems.

The mutation testing evaluations establish oracle strength as a critical evaluation dimension beyond syntactic correctness. Measuring compilation rates and test pass rates as distinct metrics reveals the layered nature of oracle quality: oracles must first compile (syntactic correctness), then execute without failing on correct code (semantic validity), and finally detect faults when present (oracle strength). This multi-level evaluation framework applies broadly to automated test generation research.

6.4 Positioning Within the Research Landscape

This work contributes to the broader neuro-symbolic AI research agenda by demonstrating concrete benefits of symbolic integration in practical software engineering tasks. While prior neuro-symbolic work focused primarily on program synthesis and code generation, this dissertation extends these techniques to the distinct problem of oracle generation where correctness requirements differ from general code generation. The findings align with results from type-constrained decoding and grammar-constrained program synthesis, confirming that formal constraints measurably improve neural generation across diverse software engineering domains.

6.5 Final Considerations

The synthesis of symbolic reasoning and neural flexibility brings both substantial benefits and notable challenges. Neuro-symbolic integration consistently improves oracle correctness across diverse experimental settings and oracle types, however it also exposes fundamental challenges: test oracle completion versus quality, flexibility versus constraint, and learned patterns versus formal guarantees. No single approach resolves these trade-offs universally: different testing contexts demand different balances.

The future does not lie in choosing exclusively between symbolic and neural paradigms, but in designing increasingly refined ways to combine them.

This dissertation shows that neuro-symbolic integration can generate semantically meaningful test oracles with stronger soundness. The empirical evidence supports the central hypothesis while also revealing the practical challenges of implementation. As language models continue to advance and symbolic analysis techniques become more sophisticated, the approaches developed here provide a foundation for next-generation testing automation that unites the strengths of both paradigms: the precision and guarantees of formal methods with the adaptability and generalization capabilities of neural models.

Bibliography

[1] Abdullin, A., Derakhshanfar, P. and Panichella, A. [2025]. Test wars: A comparative study of sbst, symbolic execution, and llm-based approaches to unit test generation, *IEEE Conference on Software Testing, Verification and Validation, ICST 2025, Napoli, Italy, March 31 - April 4, 2025*, IEEE, pp. 221–232.
URL: *https://doi.org/10.1109/ICST62969.2025.10989033*

[2] Andrews, J. H., Briand, L. C. and Labiche, Y. [2005]. Is mutation an appropriate tool for testing experiments?, *in* G. Roman, W. G. Griswold and B. Nuseibeh (eds), *27th International Conference on Software Engineering (ICSE 2005), 15-21 May 2005, St. Louis, Missouri, USA*, ACM, pp. 402–411.
URL: *https://doi.org/10.1145/1062455.1062530*

[3] Anonymous [2025]. [Replication Package] Tratto: A Neuro-Symbolic Approach to Deriving Axiomatic Test Oracles, `https://github.com/AML14/tratto-replication-package.git`.

[4] Barr, E. T., Harman, M., McMinn, P., Shahbaz, M. and Yoo, S. [2015]. The oracle problem in software testing: A survey, *IEEE Trans. Software Eng.* **41**(5): 507–525.
URL: *https://doi.org/10.1109/TSE.2014.2372785*

[5] Bavarian, M., Jun, H., Tezak, N., Schulman, J., McLeavey, C., Tworek, J. and Chen, M. [2022]. Efficient training of language models to fill in the middle, *CoRR* **abs/2207.14255**.
URL: *https://doi.org/10.48550/arXiv.2207.14255*

[6] Bekrar, S., Bekrar, C., Groz, R. and Mounier, L. [2011]. Finding software vulnerabilities by smart fuzzing, *Fourth IEEE International Conference on Software Testing, Verification and Validation, ICST 2011, Berlin, Germany, March 21-25, 2011*, IEEE Computer Society, pp. 427–430.
URL: *https://doi.org/10.1109/ICST.2011.48*

[7] Bernot, G., Gaudel, M. and Marre, B. [1991]. Software testing based on formal specifications: a theory and a tool, *Softw. Eng. J.* **6**(6): 387–405.
URL: *https://doi.org/10.1049/sej.1991.0040*

[8] Blasi, A., Goffi, A., Kuznetsov, K., Gorla, A., Ernst, M. D., Pezzè, M. and Castellanos, S. D. [2018]. Translating code comments to procedure specifications, *in* F. Tip and E. Bodden (eds), *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2018, Amsterdam, The Netherlands, July 16-21, 2018*, ACM, pp. 242–253.
URL: *https://doi.org/10.1145/3213846.3213872*

[9] Blasi, A., Gorla, A., Ernst, M. D., Pezzè, M. and Carzaniga, A. [2021]. Memo: Automatically identifying metamorphic relations in javadoc comments for test automation, *J. Syst. Softw.* **181**: 111041.
URL: *https://doi.org/10.1016/j.jss.2021.111041*

[10] Bunel, R., Hausknecht, M. J., Devlin, J., Singh, R. and Kohli, P. [2018]. Leveraging grammar and reinforcement learning for neural program synthesis, *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*, OpenReview.net.
URL: *https://openreview.net/forum?id=H1Xw62kRZ*

[11] Chan, F. T., Chen, T., Cheung, S. C., Lau, M. and Yiu, S. [1998]. Application of metamorphic testing in numerical analysis, *International Conference on Software Engineering*.
URL: *https://api.semanticscholar.org/CorpusID:59663244*

[12] Chen, Y. and Jabbarvand, R. [2024]. Neurosymbolic repair of test flakiness, *in* M. Christakis and M. Pradel (eds), *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2024, Vienna, Austria, September 16-20, 2024*, ACM, pp. 1402–1414.
URL: *https://doi.org/10.1145/3650212.3680369*

[13] Christakis, M. and Bird, C. [2016]. What developers want and need from program analysis: an empirical study, *in* D. Lo, S. Apel and S. Khurshid (eds), *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016, Singapore, September 3-7, 2016*, ACM, pp. 332–343.
URL: *https://doi.org/10.1145/2970276.2970347*

[14] Clement, C., Lu, S., Liu, X., Tufano, M., Drain, D., Duan, N., Sundaresan, N. and Svyatkovskiy, A. [2021]. Long-range modeling of source code files with eWASH: Extended window access by syntax hierarchy, *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pp. 4713–4722.

[15] Coles, H., Laurent, T., Henard, C., Papadakis, M. and Ventresque, A. [2016]. PIT: a practical mutation testing tool for java (demo), *in* A. Zeller and A. Roychoudhury (eds), *Proceedings of the 25th International Symposium on Software Testing and Analysis, ISSTA 2016, Saarbrücken, Germany, July 18-20, 2016*, ACM, pp. 449–452.
URL: *https://doi.org/10.1145/2931037.2948707*

[16] Csallner, C., Tillmann, N. and Smaragdakis, Y. [2008]. Dysy: dynamic symbolic execution for invariant inference, *in* W. Schäfer, M. B. Dwyer and V. Gruhn (eds), *30th International Conference on Software Engineering (ICSE 2008), Leipzig, Germany, May 10-18, 2008*, ACM, pp. 281–290.
URL: *https://doi.org/10.1145/1368088.1368127*

[17] Dabic, O., Aghajani, E. and Bavota, G. [2021]. Sampling projects in GitHub for MSR studies, *18th IEEE/ACM International Conference on Mining Software Repositories, MSR 2021*, IEEE, pp. 560–564.

[18] Di Grazia, L. and Pradel, M. [2023]. Code search: A survey of techniques for finding code, *ACM Comput. Surv.* **55**(11).
URL: *https://doi.org/10.1145/3565971*

[19] Dinella, E., Ryan, G., Mytkowicz, T. and Lahiri, S. K. [2022]. TOGA: A neural method for test oracle generation, *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*, ACM, pp. 2130–2141.
URL: *https://doi.org/10.1145/3510003.3510141*

[20] Ernst, M. D., Perkins, J. H., Guo, P. J., McCamant, S., Pacheco, C., Tschantz, M. S. and Xiao, C. [2007]. The daikon system for dynamic detection of likely invariants, *Sci. Comput. Program.* **69**(1-3): 35–45.
URL: *https://doi.org/10.1016/j.scico.2007.01.015*

[21] Fan, A., Gokkaya, B., Harman, M., Lyubarskiy, M., Sengupta, S., Yoo, S. and Zhang, J. M. [2023]. Large language models for software engineering: Survey and open problems, *IEEE/ACM International Conference on Software Engineering: Future of Software Engineering, ICSE-FoSE 2023, Melbourne, Australia, May 14-20, 2023*, IEEE, pp. 31–53.
URL: *https://doi.org/10.1109/ICSE-FoSE59343.2023.00008*

[22] Fielding, R. T. [2000]. *Architectural Styles and the Design of Network-based Software Architectures*, PhD thesis.

[23] Fraser, G. and Arcuri, A. [2013]. Whole test suite generation, *IEEE Trans. Software Eng.* **39**(2): 276–291.
URL: *https://doi.org/10.1109/TSE.2012.14*

[24] Fraser, G. and Arcuri, A. [2014]. A large-scale evaluation of automated unit test generation using evosuite, *ACM Trans. Softw. Eng. Methodol.* **24**(2): 8:1–8:42.
URL: *https://doi.org/10.1145/2685612*

[25] Fu, Y., Baker, E. and Chen, Y. [2024]. Constrained decoding for secure code generation, *CoRR* **abs/2405.00218**.
URL: *https://doi.org/10.48550/arXiv.2405.00218*

[26] *GitHub · Build and ship software on a single, collaborative platform — github.com* [n.d.]. https://github.com. [Accessed 28-10-2025].

[27] Goffi, A., Gorla, A., Ernst, M. D. and Pezzè, M. [2016]. Automatic generation of oracles for exceptional behaviors, *in* A. Zeller and A. Roychoudhury (eds), *Proceedings of the 25th International Symposium on Software Testing and Analysis, ISSTA 2016, Saarbrücken, Germany, July 18-20, 2016*, ACM, pp. 213–224.
URL: *https://doi.org/10.1145/2931037.2931061*

[28] Gondara, L., Simkin, J., Sayle, G., Devji, S., Arbour, G. and Ng, R. [2025]. Small or large? zero-shot or finetuned? guiding language model choice for specialized applications in healthcare, *CoRR* **abs/2504.21191**.
URL: *https://doi.org/10.48550/arXiv.2504.21191*

[29] Hoare, C. A. R. [1969]. An axiomatic basis for computer programming, *Communications of the ACM* **12**(10): 576–580.

[30] Hochreiter, S. and Schmidhuber, J. [1997]. Long short-term memory, *Neural Comput.* **9**(8): 1735–1780.
URL: *https://doi.org/10.1162/neco.1997.9.8.1735*

[31] Hossain, S. B. and Dwyer, M. B. [2025]. TOGLL: correct and strong test oracle generation with LLMS, *47th IEEE/ACM International Conference on Software Engineering, ICSE 2025, Ottawa, ON, Canada, April 26 - May 6, 2025*, IEEE, pp. 1475–1487.
URL: *https://doi.org/10.1109/ICSE55347.2025.00098*

[32] Hossain, S. B., Filieri, A., Dwyer, M. B., Elbaum, S. G. and Visser, W. [2023]. Neural-based test oracle generation: A large-scale evaluation and lessons learned, *in* S. Chandra, K. Blincoe and P. Tonella (eds), *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2023, San Francisco, CA, USA, December 3-9, 2023*, ACM, pp. 120–132.
URL: *https://doi.org/10.1145/3611643.3616265*

[33] Hossain, S. B., Taylor, R. and Dwyer, M. B. [2025]. Doc2oracll: Investigating the impact of documentation on llm-based test oracle generation, *Proc. ACM Softw. Eng.* **2**(FSE): 1870–1891.
URL: *https://doi.org/10.1145/3729354*

[34] Hosseini, P., Castro, I., Ghinassi, I. and Purver, M. [2025]. Efficient solutions for an intriguing failure of llms: Long context window does not mean llms can analyze long sequences flawlessly, *in* O. Rambow, L. Wanner, M. Apidianaki, H. Al-Khalifa, B. D. Eugenio and S. Schockaert (eds), *Proceedings of the 31st International Conference on Computational Linguistics, COLING 2025, Abu Dhabi, UAE, January 19-24, 2025*, Association for Computational Linguistics, pp. 1880–1891.
URL: *https://aclanthology.org/2025.coling-main.128/*

[35] Hou, X., Zhao, Y., Liu, Y., Yang, Z., Wang, K., Li, L., Luo, X., Lo, D., Grundy, J. and Wang, H. [2024]. Large language models for software engineering: A systematic literature review, *ACM Trans. Softw. Eng. Methodol.* **33**(8): 220:1–220:79.
URL: *https://doi.org/10.1145/3695988*

[36] Hu, E. J., Shen, Y., Wallis, P., Allen-Zhu, Z., Li, Y., Wang, S., Wang, L. and Chen, W. [2022]. Lora: Low-rank adaptation of large language models, *The Tenth International Conference on Learning Representations, ICLR 2022, Virtual Event, April 25-29, 2022*, OpenReview.net.
URL: *https://openreview.net/forum?id=nZeVKeeFYf9*

[37] Ibrahimzada, A. R., Varli, Y., Tekinoglu, D. and Jabbarvand, R. [2023]. Perfect is the enemy of test oracle, *CoRR* **abs/2302.01488**.
URL: *https://doi.org/10.48550/arXiv.2302.01488*

[38] Jain, K., Kalburgi, G. T., Le Goues, C. and Groce, A. [2023]. Mind the gap: The difference between coverage and mutation score can guide testing efforts, *34th IEEE International Symposium on Software Reliability Engineering, ISSRE 2023, Florence, Italy, October 9-12, 2023*, IEEE, pp. 102–113.
URL: *https://doi.org/10.1109/ISSRE59848.2023.00036*

[39] Johnson, B., Song, Y., Murphy-Hill, E. R. and Bowdidge, R. W. [2013]. Why don't software developers use static analysis tools to find bugs?, *in* D. Notkin, B. H. C. Cheng and K. Pohl (eds), *35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013*, IEEE Computer Society, pp. 672–681.
URL: *https://doi.org/10.1109/ICSE.2013.6606613*

[40] Just, R., Jalali, D. and Ernst, M. D. [2014]. Defects4j: a database of existing faults to enable controlled testing studies for java programs, *in* C. S. Pasareanu and D. Marinov (eds), *International Symposium on Software Testing and Analysis, ISSTA '14, San Jose, CA, USA - July 21 - 26, 2014*, ACM, pp. 437–440.
URL: *https://doi.org/10.1145/2610384.2628055*

[41] Khandaker, S. M., Kifetew, F. M., Prandi, D. and Susi, A. [2025]. Augmentest: Enhancing tests with llm-driven oracles, *IEEE Conference on Software Testing, Verification and Validation, ICST 2025, Napoli, Italy, March 31 - April 4, 2025*, IEEE, pp. 279–289.
URL: *https://doi.org/10.1109/ICST62969.2025.10988926*

[42] Konstantinou, M., Degiovanni, R. and Papadakis, M. [2024]. Do llms generate test oracles that capture the actual or the expected program behaviour?, *CoRR* **abs/2410.21136**.
URL: *https://doi.org/10.48550/arXiv.2410.21136*

[43] Lee, D. and Yannakakis, M. [1996]. Principles and methods of testing finite state machines-a survey, *Proc. IEEE* **84**(8): 1090–1123.
URL: *https://doi.org/10.1109/5.533956*

[44] Liu, N. F., Lin, K., Hewitt, J., Paranjape, A., Bevilacqua, M., Petroni, F. and Liang, P. [2024]. Lost in the middle: How language models use long contexts, *Trans. Assoc. Comput. Linguistics* **12**: 157–173.
URL: *https://doi.org/10.1162/tacl_a_00638*

[45] Lozhkov, A., Li, R., Allal, L. B., Cassano, F., Lamy-Poirier, J., Tazi, N., Tang, A., Pykhtar, D., Liu, J., Wei, Y., Liu, T., Tian, M., Kocetkov, D., Zucker, A., Belkada, Y., Wang, Z., Liu, Q., Abulkhanov, D., Paul, I., Li, Z., Li, W.-D., Risdal, M., Li, J., Zhu, J., Zhuo, T. Y., Zheltonozhskii, E., Dade, N. O. O., Yu, W., Krauß, L., Jain, N., Su, Y., He, X., Dey, M., Abati, E., Chai, Y., Muennighoff, N., Tang, X., Oblokulov, M., Akiki, C., Marone, M., Mou, C., Mishra, M., Gu, A., Hui, B., Dao, T., Zebaze, A., Dehaene, O., Patry, N., Xu, C., McAuley, J., Hu, H., Scholak, T., Paquet, S., Robinson, J., Anderson, C. J., Chapados, N., Patwary, M., Tajbakhsh, N., Jernite, Y., Ferrandis, C. M., Zhang, L., Hughes, S., Wolf, T., Guha, A., von Werra, L. and de Vries, H. [2024]. Starcoder 2 and the stack v2: The next generation.
URL: *https://arxiv.org/abs/2402.19173*

[46] Lynch, N. A. and Tuttle, M. R. [1989]. An introduction to input/output automata, *CWI quarterly* **2**: 219–246.
URL: *https://api.semanticscholar.org/CorpusID:10292247*

[47] Maj, P., Muroya, S., Siek, K., Di_Grazia, L. and Vitek, J. [2024]. The fault in our stars: Designing reproducible large-scale code analysis experiments.

[48] Meyer, B. [1992]. Applying "design by contract", *Computer* **25**(10): 40–51.
URL: *https://doi.org/10.1109/2.161279*

[49] Molina, F., Ponzio, P., Aguirre, N. and Frias, M. F. [2021]. Evospex: An evolutionary algorithm for learning postconditions, *43rd IEEE/ACM International Conference on Software Engineering, ICSE 2021, Madrid, Spain, 22-30 May 2021*, IEEE, pp. 1223–1235.
URL: *https://doi.org/10.1109/ICSE43902.2021.00112*

[50] Molinelli, D., Di Grazia, L., Martin-Lopez, A., Ernst, M. D. and Pezze, M. [2025]. Do llms generate useful test oracles? an empirical study with an unbiased dataset, *Proceedings of the 40th IEEE/ACM International Conference on Automated Software Engineering*, ASE '25.

[51] Molinelli, D., Martin-Lopez, A., Zackrone, E., Eken, B., Ernst, M. D. and Pezzè, M. [2025]. Tratto: A neuro-symbolic approach to deriving axiomatic test oracles, *CoRR* **abs/2504.04251**.
URL: *https://doi.org/10.48550/arXiv.2504.04251*

[52] Mündler, N., He, J., Wang, H., Sen, K., Song, D. and Vechev, M. T. [2025]. Type-constrained code generation with language models, *Proc. ACM Program. Lang.* **9**(PLDI): 601–626.
URL: *https://doi.org/10.1145/3729274*

[53] OpenAI [2023]. GPT-4 technical report, *CoRR* **abs/2303.08774**.
URL: *https://doi.org/10.48550/arXiv.2303.08774*

[54] Pacheco, C., Lahiri, S. K., Ernst, M. D. and Ball, T. [2007]. Feedback-directed random test generation, *29th International Conference on Software Engineering (ICSE 2007), Minneapolis, MN, USA, May 20-26, 2007*, IEEE Computer Society, pp. 75–84.
URL: *https://doi.org/10.1109/ICSE.2007.37*

[55] Pandita, R., Xiao, X., Zhong, H., Xie, T., Oney, S. and Paradkar, A. M. [2012]. Inferring method specifications from natural language API descriptions, *in* M. Glinz, G. C. Murphy and M. Pezzè (eds), *34th International Conference on Software Engineering, ICSE 2012, June 2-9, 2012, Zurich, Switzerland*, IEEE Computer Society, pp. 815–825.
URL: *https://doi.org/10.1109/ICSE.2012.6227137*

[56] Parr, T. [2013]. The definitive antlr 4 reference.

[57] Qiu, K., Di Grazia, L., Mariani, L. and Pezzè, M. [2026]. E-test: E'er-improving test suites, *International Conference on Software Engineering (ICSE)* .

[58] Richardson, L., Amundsen, M. and Ruby, S. [2013]. *RESTful Web APIs*, O'Reilly Media, Inc.

[59] Rozière, B., Gehring, J., Gloeckle, F., Sootla, S., Gat, I., Tan, X. E., Adi, Y., Liu, J., Sauvestre, R., Remez, T., Rapin, J., Kozhevnikov, A., Evtimov, I., Bitton, J., Bhatt, M., Ferrer, C. C., Grattafiori, A., Xiong, W., Défossez, A., Copet, J., Azhar, F., Touvron, H., Martin, L., Usunier, N., Scialom, T. and Synnaeve, G. [2024]. Code llama: Open foundation models for code.
URL: *https://arxiv.org/abs/2308.12950*

[60] Sadowski, C., Aftandilian, E., Eagle, A., Miller-Cushon, L. and Jaspan, C. [2018]. Lessons from building static analysis tools at google, *Commun. ACM* **61**(4): 58–66.
URL: *https://doi.org/10.1145/3188720*

[61] Tan, S. H., Marinov, D., Tan, L. and Leavens, G. T. [2012]. @tcomment: Testing javadoc comments to detect comment-code inconsistencies, *in* G. Antoniol, A. Bertolino and Y. Labiche (eds), *Fifth IEEE International Conference on Software Testing, Verification and Validation, ICST 2012, Montreal, QC, Canada, April 17-21, 2012*, IEEE Computer Society, pp. 260–269.
URL: *https://doi.org/10.1109/ICST.2012.106*

[62] Team, C., Zhao, H., Hui, J., Howland, J., Nguyen, N., Zuo, S., Hu, A., Choquette-Choo, C. A., Shen, J., Kelley, J., Bansal, K., Vilnis, L., Wirth, M., Michel, P., Choy, P., Joshi, P., Kumar, R., Hashmi, S., Agrawal, S., Gong, Z., Fine, J., Warkentin, T., Hartman, A. J., Ni, B., Korevec, K., Schaefer, K. and Huffman, S. [2024]. Codegemma: Open code models based on gemma.
URL: *https://arxiv.org/abs/2406.11409*

[63] Terragni, V., Jahangirova, G., Tonella, P. and Pezzè, M. [2020]. Evolutionary improvement of assertion oracles, *in* P. Devanbu, M. B. Cohen and T. Zimmermann (eds), *ESEC/FSE '20: 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Virtual Event, USA, November 8-13, 2020*, ACM, pp. 1178–1189.
URL: *https://doi.org/10.1145/3368089.3409758*

[64] Tomassetti, F., Smith, N., Maximilien, C. and Kirsch, S. [2021]. Javaparser.

[65] Tufano, M., Drain, D., Svyatkovskiy, A., Deng, S. K. and Sundaresan, N. [2020a]. Unit test case generation with transformers, *CoRR* **abs/2009.05617**.
URL: *https://arxiv.org/abs/2009.05617*

[66] Tufano, M., Drain, D., Svyatkovskiy, A., Deng, S. K. and Sundaresan, N. [2020b]. Unit test case generation with transformers and focal context, *CoRR* **abs/2009.05617**.
URL: *https://arxiv.org/abs/2009.05617*

[67] Tufano, M., Drain, D., Svyatkovskiy, A. and Sundaresan, N. [2022]. Generating accurate assert statements for unit test cases using pretrained transformers, *IEEE/ACM International Conference on Automation of Software Test, AST@ICSE 2022, Pittsburgh, PA, USA, May 21-22, 2022*, ACM/IEEE, pp. 54–64.
URL: *https://doi.org/10.1145/3524481.3527220*

[68] Utting, M. and Legeard, B. [2007]. *Practical Model-Based Testing - A Tools Approach*, Morgan Kaufmann.
URL: *http://www.elsevierdirect.com/product.jsp?isbn=9780123725011*

[69] Utting, M., Pretschner, A. and Legeard, B. [2012]. A taxonomy of model-based testing approaches, *Softw. Test. Verification Reliab.* **22**(5): 297–312.
URL: *https://doi.org/10.1002/stvr.456*

[70] Wang, J., Huang, Y., Chen, C., Liu, Z., Wang, S. and Wang, Q. [2024]. Software testing with large language models: Survey, landscape, and vision, *IEEE Trans. Software Eng.* **50**(4): 911–936.
URL: *https://doi.org/10.1109/TSE.2024.3368208*

[71] Wang, L., Chen, S., Jiang, L., Pan, S., Cai, R., Yang, S. and Yang, F. [2025]. Parameter-efficient fine-tuning in large language models: a survey of methodologies, *Artif. Intell. Rev.* **58**(8): 227.
URL: *https://doi.org/10.1007/s10462-025-11236-4*

[72] Watson, C., Tufano, M., Moran, K., Bavota, G. and Poshyvanyk, D. [2020]. On learning meaningful assert statements for unit test cases, *in* G. Rothermel and D. Bae (eds), *ICSE '20: 42nd International Conference on Software Engineering, Seoul, South Korea, 27 June -*

19 July, 2020, ACM, pp. 1398–1409.
URL: *https://doi.org/10.1145/3377811.3380429*

[73] Wei, Y., Furia, C. A., Kazmin, N. and Meyer, B. [2011]. Inferring better contracts, *in* R. N. Taylor, H. C. Gall and N. Medvidovic (eds), *Proceedings of the 33rd International Conference on Software Engineering, ICSE 2011, Waikiki, Honolulu , HI, USA, May 21-28, 2011*, ACM, pp. 191–200.
URL: *https://doi.org/10.1145/1985793.1985820*

[74] Xu, R., Wang, Z., Fan, R. and Liu, P. [2024]. Benchmarking benchmark leakage in large language models, *CoRR* **abs/2404.18824**.
URL: *https://doi.org/10.48550/arXiv.2404.18824*

[75] Yang, A., Yang, B., Zhang, B., Hui, B., Zheng, B., Yu, B., Li, C., Liu, D., Huang, F., Wei, H., Lin, H., Yang, J., Tu, J., Zhang, J., Yang, J., Yang, J., Zhou, J., Lin, J., Dang, K., Lu, K., Bao, K., Yang, K., Yu, L., Li, M., Xue, M., Zhang, P., Zhu, Q., Men, R., Lin, R., Li, T., Tang, T., Xia, T., Ren, X., Ren, X., Fan, Y., Su, Y., Zhang, Y., Wan, Y., Liu, Y., Cui, Z., Zhang, Z. and Qiu, Z. [2025]. Qwen2.5 technical report, https://arxiv.org/abs/2412.15115.

[76] Yoo, S. and Harman, M. [2012]. Regression testing minimization, selection and prioritization: a survey, *Software testing, verification and reliability* **22**(2): 67–120.

[77] Zhou, K., Zhu, Y., Chen, Z., Chen, W., Zhao, W. X., Chen, X., Lin, Y., Wen, J. and Han, J. [2023]. Don't make your LLM an evaluation benchmark cheater, *CoRR* **abs/2311.01964**.
URL: *https://doi.org/10.48550/arXiv.2311.01964*