Generazione Automatica di Filter Code per IFTTT tramite Large Language Models

Corso di Intelligenza Artificiale
Progetto su Natural Language Processing
Prof. Vincenzo Deufemia

Studenti: Mario Balbi, Francesco De Stasio, Antonio Imperiale

Abstract

Questo progetto esplora l'applicazione dei Large Language Models (LLM) per automatizzare la generazione di codice JavaScript, noto come *"filter code"*, per la piattaforma IFTTT. La scrittura manuale di questo codice rappresenta una barriera significativa per gli utenti non tecnici. Per affrontare questa sfida, è stato sviluppato un sistema che orchestra un processo a più fasi: un crawler raccoglie dati dettagliati sui servizi IFTTT, un generatore costruisce un dataset strutturato e, infine, un LLM impiega una strategia di prompting a due fasi. La prima fase genera un *"intent"* specifico dalla descrizione in linguaggio naturale dell'utente; la seconda fase traduce questo intent in codice. La valutazione quantitativa su 563 campioni mostra un tasso di successo del 53.3% nella generazione di intent validi, dimostrando la fattibilità dell'approccio. Uno studio comparativo ha inoltre validato la superiorità della strategia a due fasi rispetto a un approccio diretto a singola fase, che risulta significativamente più propenso a errori. Il lavoro si conclude con un'analisi delle considerazioni etiche e di sicurezza legate alla generazione automatica di codice. Questo studio non solo dimostra la fattibilità di tradurre le intenzioni umane in codice eseguibile, ma ne analizza criticamente l'architettura e le implicazioni.

1 Introduzione

Nel campo dell'End-User Development (EUD) [1, 2, 3], le piattaforme Trigger-Action (TAP) come IFTTT (If This Then That) hanno democratizzato l'automazione, consentendo agli utenti di creare catene di istruzioni condizionali, chiamate *"applet"* [4]. Un'applet connette due o più servizi per eseguire un'azione automatica in risposta a un trigger (es. "Se ricevo un'email con allegato, allora salvalo su Dropbox").

Per automazioni più sofisticate, IFTTT offre il *"filter code"*, che permette di scrivere logica personalizzata in JavaScript per controllare il comportamento di un'applet [5]. Questa funzionalità espone un ambiente di esecuzione in cui i dati del trigger ("ingredients") sono disponibili come variabili e le azioni possono essere manipolate tramite metodi specifici (es. ‘skip()‘). Tuttavia, ciò richiede competenze di programmazione, limitando l'adozione da parte di utenti non tecnici.

L'avvento dei Large Language Models (LLM) ha aperto nuove frontiere nella generazione di codice da linguaggio naturale [6, 7]. Questo progetto sfrutta tale capacità per abbattere la barriera tecnica della scrittura di filter code. L'obiettivo è creare un sistema end-to-end che traduca una descrizione in linguaggio naturale in codice JavaScript valido e funzionante per IFTTT, un problema affrontato anche da studi recenti che mirano a generare componenti di regole da descrizioni in linguaggio naturale [8, 9].

2 Stato dell'Arte

La ricerca ha esplorato vari metodi per migliorare l'usabilità delle TAP. Approcci iniziali si sono concentrati sulla semplificazione della creazione di regole tramite sistemi visuali [10] o programmazione end-user [11, 1], che però richiedono ancora una configurazione manuale dei componenti, limitando flessibilità e scalabilità.

Più di recente, l'applicazione di tecniche di Natural Language Processing (NLP) ha permesso di generare componenti di regole da descrizioni testuali [8, 12], facilitando la configurazione ma senza affrontare la personalizzazione del comportamento tramite codice. Altri studi si sono focalizzati sul debugging e sulla prevenzione di comportamenti inattesi, raccomandando modifiche per correggere le regole [13, 14, 15] o identificando violazioni di sicurezza e privacy [16, 17, 18]. Tuttavia, questi approcci non consentono agli utenti di sfruttare appieno le capacità della piattaforma tramite codice personalizzato.

Il nostro lavoro si inserisce nel campo della generazione di codice da linguaggio naturale, un'area che ha visto enormi progressi con l'avvento dei modelli transformer [7]. Mentre i primi metodi si basavano su template [19] o regole sintattiche [20], i modelli sequence-to-sequence, originariamente per la traduzione automatica, sono stati adattati per generare codice [21, 22]. Modelli più recenti come OpenAI Codex [6] e CodeT5 [23], pre-addestrati su vasti repository di codice, hanno mostrato miglioramenti significativi.

3 Obiettivi del Progetto

Gli obiettivi principali di questo studio sono stati definiti per affrontare sistematicamente il problema, dalla raccolta dei dati alla generazione del codice:

- Sviluppare un sistema di web crawling robusto, basato su Selenium, per estrarre in modo completo e strutturato i metadati dei servizi, trigger e azioni dalla piattaforma IFTTT, inclusi dettagli tecnici come "slug", "ingredients" e metodi specifici del filter code.
- Costruire un dataset di alta qualità in formato JSON Lines, dove ogni record associ una descrizione in linguaggio naturale di un'automazione a tutti i dettagli tecnici necessari per la generazione del codice, ottenuti tramite la fase di crawling.
- Progettare e implementare una pipeline basata su LLM per la generazione di codice, introducendo una strategia a due fasi (generazione di "intent" e successiva generazione di codice) per migliorare la precisione, la specificità e l'affidabilità del risultato finale.

- Dimostrare la fattibilità e l'efficacia dell'approccio attraverso un'analisi qualitativa dettagliata degli output generati dal sistema, verificando la coerenza tra l'intento dell'utente, l'intent intermedio e il filter code finale.

4 Architettura e Implementazione

Il sistema è stato progettato con un'architettura modulare che riflette le tre fasi principali del progetto.

4.1 Dettagli Implementativi

Il sistema è stato sviluppato in Python 3.12. Per l'automazione del browser e il web crawling, è stata utilizzata la libreria Selenium 4, in combinazione con 'webdriver-manager' per la gestione automatica dei driver. La manipolazione dei dati è stata affidata alla libreria Pandas.

Per l'interazione con i modelli linguistici, è stata impiegata la libreria 'openai', configurata per comunicare con un server di inferenza locale compatibile con l'API OpenAI (es. LM Studio). Il modello primario utilizzato per la generazione sia dell'intent che del filter code è stato 'meta-llama/Llama-3-70B-Instruct', scelto per il suo ottimo bilanciamento tra performance e capacità di seguire istruzioni complesse.

4.2 Data Collection

La prima fase consiste nel raccogliere informazioni dettagliate da IFTTT. Questo compito è affidato a una serie di script nella directory crawler/.

- **Scraping dei Servizi:** Utilizzando `Selenium`, lo script crawler.py simula la navigazione di un utente per estrarre un elenco completo di tutti i servizi disponibili e i loro URL.
- **Estrazione dei Dettagli:** Per ogni servizio, il sistema visita il suo URL e, tramite funzioni come get_triggers_actions_queries, identifica i link a tutti i trigger, azioni e query associati. Successivamente, la funzione extract_detail_data analizza ciascuno di questi link per estrarre metadati cruciali: **Developer Info** (es. API endpoint slug), **Trigger/Action Fields** (campi di configurazione), e **Ingredients** (le variabili e la loro sintassi di accesso nel codice).
- **Esecuzione Parallela:** Lo script crawler_saver.py impiega un `ThreadPoolExecutor` per parallelizzare il processo di scraping, ottimizzando i tempi di raccolta e salvando i dati in un unico file JSON).

4.3 Generazione del Dataset

Gli script nella directory generate/ sono responsabili della trasformazione dei dati grezzi in un dataset raffinato, pronto per essere utilizzato dal LLM. La base di partenza è il dataset pubblico "IFTTT Recipes" [24]. Questo dataset, contenente oltre 50.000 "ricette" (applet), fu originariamente creato e presentato al workshop SenSys '21 per studiare come gli esseri umani utilizzano i dispositivi Internet-of-Things (IoT) e quali comportamenti si aspettano da essi.

Per costruire la nostra base di dati iniziale, sono stati aggregati tutti i file principali del dataset (da 'Step1' a 'Step4'), in particolare per consolidare un insieme di applet che includessero esempi reali di 'filter_code' scritti dagli utenti. Da questa aggregazione è risultato un dataset preliminare di circa 3500 record. Successivamente, una fase di pulizia ha rimosso i duplicati e le righe con valori nulli o incompleti, riducendo il dataset finale a 1740 record validi. Il processo di arricchimento è quindi proseguito: per ogni riga di questa base dati pulita, il sistema identifica il trigger e l'azione corrispondenti e li associa ai metadati dettagliati ottenuti nella fase di crawling. Questo produce un file JSON Lines (.jsonl), un formato ideale per l'elaborazione in streaming da parte dei modelli linguistici. Ogni riga del file finale rappresenta un'automazione completa e autoconsistente, contenente tutte le informazioni necessarie per il task generativo: la descrizione originale, i dettagli completi del trigger (canale, slug, campi, ingredienti con la loro sintassi) e i dettagli dell'azione.

4.4 Progettazione dell'Approccio Generativo

Durante lo sviluppo, è stato inizialmente considerato un approccio a singola fase (descrizione -> codice). Per testarne la fattibilità, è stato configurato un task di generazione in cui al LLM ('meta-llama/Llama-3-8B-Instruct') veniva chiesto di generare direttamente il 'filter_code' a partire dalla 'original_description' e dai metadati.

I risultati, tuttavia, si sono rivelati largamente insoddisfacenti. Il modello ha mostrato una tendenza significativamente maggiore a:

- **Produrre Codice Incompleto o Generico:** Spesso il codice generato ometteva la logica condizionale, limitandosi a impostare parametri dell'azione senza un vero controllo.
- **Allucinare 'ingredients':** Il modello inventava nomi di variabili non esistenti, tentando di indovinare la sintassi corretta.
- **Fraintendere la Logica:** L'ambiguità della descrizione originale si traduceva direttamente in codice errato. Ad esempio, una richiesta come "se la temperatura è bassa" veniva ignorata o tradotta con valori di soglia arbitrari e spesso inadeguati.

Qualitativamente, il tasso di successo nella generazione di codice corretto e funzionale è stato stimato inferiore al 20%. Questi scarsi risultati hanno reso evidente la necessità di una strategia più strutturata, motivando la progettazione della pipeline a due fasi descritta di seguito.

4.5 Generazione del Codice con l'Approccio a Due Fasi

I fallimenti dell'approccio diretto hanno portato allo sviluppo di una strategia a due passaggi per massimizzare la qualità del codice. Questa è la fase centrale del sistema, implementata nella directory llm/, e si articola come segue:

(1) **Generazione dell'Intent:** Lo script generate_intent_from_jsonl.py invia a un LLM una riga del dataset. Il prompt di sistema (system_prompt_generate_intent.txt) istruisce il modello a trasformare la original_description in un **intent** preciso e implementabile, risolvendo le ambiguità (es. trasformando "se fa freddo" in "se la temperatura scende sotto i 5 gradi Celsius").
(2) **Generazione del Filter Code:** Ottenuto un intent chiaro, lo script generate_filtercode_from_intent.py esegue il secondo passaggio. Invia all'LLM la stessa struttura dati arricchita con l'intent. Il prompt di sistema (system_prompt_

generate_filtercode_from_intent.txt) istruisce il modello a ignorare la descrizione originale e a basarsi **esclusivamente sull'intent** per produrre il codice JavaScript, usando gli "ingredients" e i metodi dell'azione.

È importante sottolineare che l'approccio di questo progetto si basa sul *prompt engineering* e sull'*in-context learning*, differenziandosi da altre ricerche come [25] che impiegano il *fine-tuning*. Questa scelta è stata deliberata: permette di sfruttare le potenti capacità zero-shot di modelli generalisti di grandi dimensioni senza la necessità di sostenere i costi computazionali e i tempi richiesti per il riaddestramento, offrendo al contempo maggiore flessibilità nel testare diversi modelli.

5 Sperimentazione e Risultati

Per valutare l'efficacia del sistema, sono state condotte diverse analisi quantitative e qualitative, i cui risultati sono presentati di seguito.

5.1 Valutazione Quantitativa dell'Intent

Per integrare l'analisi qualitativa, è stata condotta una valutazione quantitativa sulla fase di generazione degli intent, che rappresenta il primo e più critico passaggio del nostro approccio a due stadi.

A partire dal dataset arricchito, sono stati generati un totale di 1740 intent utilizzando il modello 'meta-llama/Llama-3-70B-Instruct'. Successivamente, è stata eseguita una validazione manuale su un campione casuale di 563 di questi intent generati. Durante questa fase, sono stati scartati preliminarmente i campioni la cui 'original_description' conteneva un mix di più lingue, per garantire l'omogeneità del campione. La validazione successiva consisteva nel verificare se l'intent fosse:

- **Chiaro e non ambiguo:** L'intent doveva risolvere le genericità della descrizione originale.
- **Implementabile:** La logica descritta doveva essere traducibile in codice utilizzando gli 'ingredients' e i metodi disponibili.
- **Corretto:** L'intent doveva riflettere fedelmente l'obiettivo dell'utente descritto nel testo originale.

I risultati di questa validazione manuale sono i seguenti:

- **Intent Validi:** 300 su 563
- **Percentuale di Successo:** 53.3%

Questo risultato, sebbene non perfetto, è estremamente promettente. Dimostra che un LLM generalista, senza alcun fine-tuning, è in grado di interpretare correttamente e trasformare una descrizione utente in un piano strutturato e implementabile in più della metà dei casi. Questo dato convalida l'efficacia della nostra strategia a due fasi, in quanto un intent di alta qualità è il prerequisito fondamentale per una generazione di codice di successo.

5.2 Analisi Comparativa della Generazione di Codice

Una volta validata l'efficacia della generazione dell'intent, è stata condotta un'analisi quantitativa per confrontare le performance dei due modelli LLM utilizzati per la seconda fase: la generazione del 'filter_code' a partire dall'intent. I due modelli, 'meta-llama/Llama-3-70B-Instruct' e 'Qwen/Qwen3-32B', sono stati incaricati di generare il codice per tutti i 300 intent validati manualmente.

La metrica di valutazione scelta è stata la **correttezza sintattica** del codice JavaScript prodotto. Ogni blocco di codice generato è stato analizzato programmaticamente tramite il parser JavaScript 'esprima' per verificare l'assenza di errori di sintassi. I risultati di questo confronto sono riportati in Tabella 1.

Modello	Codici Corretti	Tasso di Successo
Llama-3-70B-Instruct	296 / 300	98.67%
Qwen3-32B	288 / 300	96.00%

Table 1: Confronto della correttezza sintattica del codice generato dai due modelli.

Entrambi i modelli dimostrano una capacità estremamente elevata di produrre codice sintatticamente valido quando partono da un intent chiaro e strutturato, con tassi di successo superiori al 95%. Questo risultato è notevole e conferma la robustezza dell'approccio a due fasi. Si osserva un leggero vantaggio per 'Llama-3-70B-Instruct', che raggiunge il 98.67% di correttezza, rafforzando l'osservazione qualitativa precedentemente riportata riguardo alla sua maggiore affidabilità nella gestione di logiche complesse.

5.3 Analisi degli Errori

Per comprendere meglio le cause dei fallimenti nella generazione dell'intent, è stata condotta un'analisi sui 263 esempi del campione di validazione che sono stati etichettati come non validi. Utilizzando la rubrica di validità definita, gli errori sono stati classificati in categorie distinte. La distribuzione degli errori è presentata in Tabella 2.

Categoria di Errore	Conteggio	Percentuale
Logica Fraintesa o Allucinazione	260	98.86%
Descrizione Vuota	2	0.76%
Intent Incompleto o Vuoto	1	0.38%

Table 2: Distribuzione delle categorie di errore negli intent non validi.

Il risultato più evidente è che la quasi totalità degli errori (98.86%) ricade nella categoria più complessa di **Logica Fraintesa o Allucinazione**. Questo indica che il modello raramente commette errori banali, come generare un intent vuoto o fallire in presenza di una descrizione vuota. La sfida principale risiede nella corretta interpretazione semantica della richiesta dell'utente e nella sua mappatura sulle funzionalità esposte dai metadati, senza inventare parametri o fraintendere la condizione logica desiderata. Rientrano in questa categoria anche i casi in cui la descrizione originale conteneva un mix di più lingue, portando il modello a generare un intent confuso o semanticamente incoerente.

Questo dato rafforza ulteriormente la necessità di un approccio strutturato come la nostra pipeline a due fasi. Un singolo passaggio diretto da descrizione a codice sarebbe ancora più esposto a questo tipo di fallimenti semantici. La generazione di un 'intent' intermedio, anche se fallisce in circa il 47% dei casi, agisce come un filtro critico

che semplifica il compito successivo e permette di isolare e, in futuro, mitigare questa classe di errori complessi.

5.4 Discussione e Limiti

Nonostante i risultati promettenti, il sistema presenta alcune limitazioni. L'analisi qualitativa e quantitativa ha rivelato che il modello LLM incontra difficoltà in scenari di elevata complessità logica, come la gestione di multiple condizioni booleane annidate. Inoltre, la qualità della generazione è risultata dipendente dalla ricchezza dei metadati del servizio IFTTT: servizi meno documentati hanno portato a un maggior numero di errori.

Una significativa limitazione metodologica riguarda la valutazione del codice generato. Non è stato possibile eseguire test funzionali diretti sulla piattaforma IFTTT, poiché ogni 'filter code' può richiedere l'accesso a servizi eterogenei, account utente specifici e, in alcuni casi, dispositivi hardware dedicati. Di conseguenza, la valutazione si è concentrata sulla correttezza sintattica e sulla coerenza logica con l'intent, piuttosto che sull'esecuzione effettiva.

6 Considerazioni Etiche e di Sicurezza

L'automazione della generazione di codice solleva questioni critiche. Sul piano **etico**, esiste il rischio di abusi (es. spam, disinformazione). Le mitigazioni includono il filtraggio di intent dannosi, la limitazione delle funzionalità sensibili e la trasparenza verso l'utente, che deve essere incoraggiato a revisionare il codice. Sul piano della **sicurezza**, il codice generato potrebbe essere insicuro, ad esempio esponendo dati sensibili. Per contrastare ciò, è fondamentale integrare analisi di sicurezza automatiche (SAST, taint analysis) e istruire il modello a seguire pratiche di codifica sicure by-design.

7 Conclusioni e Sviluppi Futuri

In conclusione, questo progetto ha dimostrato la fattibilità della generazione automatica di 'filter code' per IFTTT tramite LLM, validando un'architettura a due fasi superiore a un approccio diretto. Con un successo del 53.3% nella generazione di intent, il sistema è un passo promettente verso una maggiore accessibilità dell'automazione.

Gli **sviluppi futuri** si concentreranno su quattro aree principali: la creazione di un framework di valutazione automatica per test funzionali; l'implementazione di contromisure di sicurezza (es. SAST); il fine-tuning di modelli specifici sul nostro dataset per migliorare l'accuratezza; e lo sviluppo di un'interfaccia utente interattiva con un ciclo di feedback per un miglioramento continuo del sistema.

References

[1] H. Lieberman, F. Paternò, M. Klann, and V. Wulf. 2006. End-user development: an emerging paradigm. In *End User Development*. Springer, 1–8.

[2] F. Paternò. 2013. End user development: survey of an emerging field for empowering people. *Int. Scholarly Res. Not.*, 2013, 532659.

[3] B.R. Barricelli, F. Cassano, D. Fogli, and A. Piccinno. 2019. End-user development, end-user programming and end-user software engineering: a systematic mapping study. *J. Syst. Softw.*, 149, 101–137.

[4] B. Ur, E. McManus, Y.H. Pak, and M.L. Littman. 2014. Practical trigger-action programming in the smart home. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, 803–812.

[5] Y. Chen, M. Alhanahnah, A. Sabelfeld, R. Chatterjee, and E. Fernandes. 2022. Practical data access minimization in trigger-action platforms. In *Proceedings of 31st USENIX Security Symposium*, 2929–2945.

[6] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Igor Mordatch, Maciej Chociej, Jie Tang, Mo Bavarian, Christina Jang, Chris Hesse, Mark Ning, Siqi Ouyang, Chris Brundage, Mira Murati, Ilya Sutskever, and Wojciech Zaremba. 2021. Evaluating large language models trained on code. (2021). arXiv: 2107.03374 [cs.LG].

[7] Junda Jiang, Feiyu Wang, Jialun Shen, Sung-Hwan Kim, and Seok-Won Kim. 2024. A survey on large language models for code generation. (2024). arXiv: 2406.00515 [cs.SE].

[8] Y. Gao, K. Xiao, F. Li, W. Xu, J. Huang, and W. Dong. 2024. Chatiot: zero-code generation of trigger-action based iot programs. *Proc. ACM Interact. Mob. Wearable Ubiquit. Technol.*, 8, 3, 1–29.

[9] I.N.B. Yusuf, D.B.A. Jamal, L. Jiang, and D. Lo. 2022. Accurate generation of trigger-action programs with domain-adapted sequence-to-sequence learning. In *Proceedings of the IEEE/ACM International Conference on Program Comprehension*, 99–110.

[10] L. Corcella, M. Manca, F. Paternò, and C. Santoro. 2019. A visual tool for analysing iot trigger/action programming. In *Proceedings of International Working Conference Human-Centered Software Engineering*. Springer, 189–206.

[11] A. Krishna, M. Le Pallec, R. Mateescu, and G. Salaün. 2021. Design and deployment of expressive and correct web of things applications. *ACM Trans. Internet Things*, 3, 1, 1–30.

[12] I.N.B. Yusuf, D.B.A. Jamal, L. Jiang, and D. Lo. 2022. Recipegen++: an automated trigger action programs generator. In *Proceedings of the ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 1672–1676.

[13] F. Corno, L. De Russis, and A. Monge Roffarello. 2019. Empowering end users in debugging trigger-action rules. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*, 1–13.

[14] L. Zhang, W. He, J. Martinez, N. Brackenbury, S. Lu, and B. Ur. 2019. Autotap: synthesizing and repairing trigger-action programs using ltl properties. In *Proceedings of IEEE/ACM 41st International Conference on Software Engineering*, 281–291.

[15] L. Zhang, C. Zhou, M.L. Littman, B. Ur, and S. Lu. 2023. Helping users debug trigger-action programs. *Proc. ACM Interact. Mob. Wearable Ubiquit. Technol.*, 6, 4, 1–32.

[16] B. Breve, G. Cimino, and V. Deufemia. 2022. Identifying security and privacy violation rules in trigger-action iot platforms with nlp models. *IEEE Internet Things J.*, 10, 6, 5607–5622.

[17] B. Breve, G. Cimino, G. Desolda, V. Deufemia, and A. Elefante. 2023. On the user perception of security risks of tap rules: a user study. In *Proceedings of International Symposium on End User Development (IS-EUD 2023)* (Lecture Notes in Computer Science). Volume 13917. Springer, 162–179.

[18] B. Breve, G. Cimino, and V. Deufemia. 2024. Hybrid prompt learning for generating justifications of security risks in automation rules. *ACM Trans. Intell. Syst. Technol.*, 15, 5, 1–26.

[19] K. Hu, Z. Duan, J. Wang, L. Gao, and L. Shang. 2019. Template-based aadl automatic code generation. *Front. Comput. Sci.*, 13, 698–714.

[20] I.S. Bajwa, M.I. Siddique, and M.A. Choudhary. 2006. Rule based production systems for automatic code generation in java. In *Proceedings of 1st International Conference on Digital Information Management*, 300–305.

[21] W. Ling, E. Grefenstette, K.M. Hermann, T. Kočiský, A. Senior, F. Wang, and P. Blunsom. 2016. Latent predictor networks for code generation. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, 599–609.

[22] Pengcheng Yin and Graham Neubig. 2017. A syntactic neural model for general-purpose code generation. (2017). arXiv: 1704.01696 [cs.CL].

[23] Y. Wang, W. Wang, S. Joty, and S.C.H. Hoi. 2021. Codet5: identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, 8696–8708.

[24] Haoxiang Yu, Jie Hua, and Christine Julien. 2021. Dataset: analysis of ifttt recipes to study how humans use internet-of-things (iot) devices. In *Proceedings of the 19th ACM Conference on Embedded Networked Sensor Systems*. Association for Computing Machinery, 537–541. DOI: 10.1145/3485730.3494115.

[25] G. Cimino and V. Deufemia. 2025. Sigfrid: unsupervised, platform-agnostic interference detection in iot automation rules. *ACM Trans. Internet Things*.