Online-Journals.org

iJIM International Journal of Interactive Mobile Technologies

iJIM | eISSN: 1865-7923 | Vol. 20 No. 3 (2026) | OPEN ACCESS

https://doi.org/10.3991/ijim.v20i03.59861

PAPER

Large Language Model Selection for Test-Driven Prompt Android iOS Development

Muhammad Rizqullah (✉), Emad Albassam

King Abdulaziz University, Jeddah, Saudi Arabia

mrizqullah@stu.kau.edu.sa

ABSTRACT

Large language model (LLM) code generation research predominantly focuses on Python, with test-driven prompt engineering exclusively targeting this language. This study presents a comprehensive LLM selection framework for mobile development through rigorous empirical analysis. We conducted 8,704 evaluations across 544 programming tasks (HumanEval and MBPP datasets) on Android (Java) and iOS (Swift) platforms using four state-of-the-art LLMs (GPT-4o, GPT-4o-mini, Qwen 14B, and Qwen 32B), two prompting strategies (base and test-driven), and two metrics (accuracy and remediation accuracy). Systematic analysis of platform-specific patterns yielded a decision tree incorporating first-attempt correctness, budget constraints, and self-hosting requirements, validated through three industry-relevant use cases. Results show test-driven prompting (TDP) achieves a +2.22 pp average accuracy improvement over baseline (95% CI [1.22–3.23 pp], $p < 0.001$, $d = 0.3974$). However, LLMs consistently underperform in mobile development (66.85%–88.87%) compared to Python-based code generation (86.90%–91.30%) regardless of model size or type. This framework establishes groundwork for platform-specific optimizations while providing practitioners with actionable guidance for model selection in mobile development contexts.

KEYWORDS

artificial intelligence (AI), explainable AI, empirical software engineering, mobile development, software engineering

1 INTRODUCTION

Through programs such as GitHub Copilot and ChatGPT, large language models (LLMs) have revolutionized software development [1], [2], significantly lowering development effort and allowing non-programmers to quickly prototype applications for market validation [3]. However, due to their limited explainability, LLMs produce believable but factually incorrect text, which is referred to as hallucinations [4], [5]. Using previous context, LLMs generate text through probability-based token prediction [6], [7], resulting in syntactically correct but semantically flawed

Rizqullah, M., Albassam, E. (2026). Large Language Model Selection for Test-Driven Prompt Android iOS Development. *International Journal of Interactive Mobile Technologies (iJIM)*, 20(3), pp. 71–82. https://doi.org/10.3991/ijim.v20i03.59861

Article submitted 2025-09-14. Revision uploaded 2025-11-11. Final acceptance 2025-11-16.

code [2], [8]. As a result, LLM-generated code might have minor mistakes that need careful testing and expert review [9], [10].

A systematic stage of development, software testing verifies that code satisfies intended requirements and specifications. Costs are significantly increased when defects reach end users rather than being discovered during development [11], [12], especially in mobile development. Mobile applications are subject to immediate, public quality evaluation through platform-specific rating systems on the Google Play Store and Apple App Store [14], in contrast to web applications, where user feedback may be indirect [13]. Defects and performance problems are instantly revealed by user ratings and reviews, generating an open quality signal that has a direct influence on the adoption and reputation of applications [15]. Errors in mobile environments therefore have higher costs, including not only repairs but also possible revenue loss and reputational harm due to negative public perception. By mandating that developers create automated, requirements-compliant tests prior to deploying functional code, test-driven development (TDD) successfully reduces these risks [16], [17]. By forcing developers to carefully consider requirements and system design before coding, this method lowers production defects. Expanding on this strategy, researchers have looked into test-driven prompting (TDP), a prompt engineering technique that incorporates test suites that record software requirements straight into prompts. Without requiring extra development work, this technique increases the accuracy of LLM-generated code and permits automatic validation against the embedded tests [18], [19], [20], and [21]. However, there are still a lot of unanswered questions regarding LLM performance for platform-specific development, especially mobile applications on the iOS (Swift) and Android (Java) platforms, as research has mostly concentrated on general-purpose languages such as Python [22], [23]. Even simple mobile programming tasks are not adequately addressed by current LLM evaluation frameworks [24], [25], and platform-specific issues such as resource limitations, specialized APIs, and stringent compilation requirements are mostly disregarded [26], [27]. Since TDP has only been studied in Python environments, these limitations are especially noticeable for it [18], [19], [20], and [21]. There is no empirical data on TDP's efficacy in mobile development contexts, despite the fact that mobile platforms are more susceptible to software bugs and thus require error reduction [28], [29], and [30].

Three research questions are addressed in this study in order to increase understanding of LLM prompt engineering techniques in mobile development. First, when using TDP for Android (Java) development, which LLM produces the most accurate code? Second, which LLM works best with TDP for iOS (Swift) development? Third, how can developers choose models that best fit the target platform, accuracy, cost, and self-hostability? We answer these questions to ascertain whether TDP results in statistically significant improvements in mobile development and to give developers a methodical framework for choosing models that strike a balance between deployment constraints, accuracy requirements, and computational budget. The comprehensive experimental data presented in this study, including all datasets and results, are openly accessible via the Zenodo data repository at https://doi.org/10.5281/zenodo.17735859.

2 MATERIALS AND METHODS

Our study focuses on finding the best LLMs to use TDP on the mobile platform described in Figure 1. The following subsections will explain each step in detail.

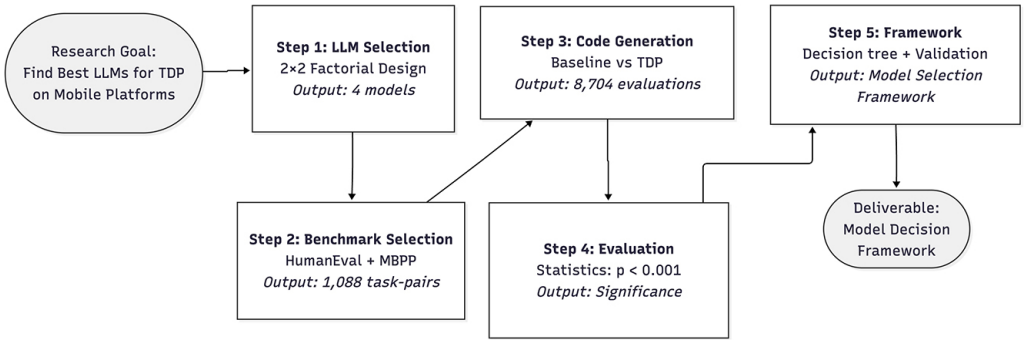

Fig. 1. Overview of the five-step research methodology for identifying optimal LLMs for TDP on a mobile platform

2.1 LLM selection

We selected models using a 2×2 factorial design spanning two critical dimensions for developers to consider when choosing an LLM as presented in Table 1.

Table 1. Selected LLMs for 2×2 factorial design across model size and type

Model	Size	Type
GPT4o	Large	Proprietary
GPT4omini	Small	Proprietary
Qwen 2.5 Coder 32B	Large	Open Source
Qwen 2.5 Coder 14B	Small	Open Source

This design allows us to analyze main effects (size, type) and interactions, providing generalizable insights beyond the specific models tested. The models are chosen based on research that comprehensively selects the best models for the benchmarks we are using below [9]. The following are the costs of the LLMs from the API providers we use: 1) GPT4o: $12.5 USD per input-output combined million tokens, 2) GPT4omini: combined $0.75 USD per input-output combined million tokens, 3) Qwen 2.5 Coder 32B: $5 per hour hosted on a computing node with 2 Nvidia A100 GPUs 160 GB, 4) Qwen 2.5 Coder 14B: $1.8 USD per hour hosted on a computing node with 1 Nvidia L40S 48 GB GPU.

2.2 Benchmark selection

To evaluate LLM performance using TDP, we employ two established benchmarks previously translated to Java (Android) and Swift (iOS) [22]: 1) HumanEval: comprehensive unit tests are used to test data structures, algorithms, and edge cases in 158 algorithmic problems per language (316 total). 2) MBPP [22]: 386 programming tasks for each language (772 total) that include assertion-based tests for data transformations, text processing, list operations, and numerical computations. The only difference between the two benchmarks' problems is their syntax, which isolates performance variations to the LLMs themselves. Our benchmarks assess fundamental programming concepts that mobile developers use on a daily basis, even though they do not include framework-specific APIs. For example,

properly implementing array filtering in a benchmark corresponds to filtering search results in RecyclerView or UITableView implementations.

2.3 LLM code generation

We use a 2×2 factorial design in two dimensions to process benchmark problems into prompts: (1) Prompt techniques: baseline (problem description only) with TDP, which uses the first 50% of test cases as guardrails; and (2) Number of attempts: one attempt with multiple attempts at remediation (where error feedback directs subsequent prompts, with a maximum of five attempts based on previous work [20] demonstrating diminishing returns). Diverse developer practices in LLM-based mobile development are captured in this design. For every attempt at code generation, we carry out the following procedures to guarantee repeatability:

1. Have a standardized single system prompt that is separated and invisible from the prompting techniques to ensure output consistency.
2. Have a single set of hyperparameters for all code generation attempts that are focused on ensuring determinism: a) Temperature: 0 [20], b) Seed: 1000 [20] c) Max token: Default model limits.
3. Large language model access: We use LLMs through two API providers that are versioned to prevent updates to the models affecting future code generation experiments: a) Hugging Face Inference Endpoints for Qwen models (Qwen/Qwen2.5-Coder-32B-Instruct, Qwen/Qwen2.5-Coder-14B-Instruct), b) OpenRouter API for GPT models (openai/gpt-4o-2024-11-20, openai/gpt-4o-mini-2024-07-18)
4. Documented code dependency: Our experiments are automated using Python 3.11.13. Besides standard libraries, here are the libraries we use: a) numpy = "^2.3.1", b) pandas = "^2.3.0", c) openai = "^1.93.0", d) python-dotenv = "^1.1.1", e) scipy = "^1.16.1", f) pingouin = "^0.5.5"

2.4 Code evaluation

Since it's possible that mobile developers use the first AI suggestions without coming up with alternatives, we report pass@1 accuracy (the proportion of issues resolved on the first attempt). Additionally, we provide remediation accuracy, which is the percentage of problems resolved in five tries with iterative error feedback as the alternative scenario. We used Shapiro-Wilk tests for normality; for non-normal data, we use Wilcoxon signed-rank tests; for normal data, we use paired t-tests. To determine the statistical significance of TDP versus the baseline prompt ($\alpha = 0.05$). Cohen's d was used to calculate effect sizes (negligible: $|d| < 0.2$; small: $0.2 \leq |d| < 0.5$; medium: $0.5 \leq |d| < 0.8$; large: $|d| \geq 0.8$). The t-distribution method is used to report mean differences along with 95% confidence intervals.

2.5 Decision framework construction

Larger, more costly models are generally thought to perform better than smaller, more affordable models on all tasks. We divided the LLM results by platform (iOS and Android) and determined the average accuracy for each in order to test this assumption in mobile development. Three lenses were used to systematically analyze the results: (1) model size (large/small), (2) model type (proprietary/open source),

and (3) cost per token. We used these factors as sequential decision nodes in an evidence-based decision tree that we built based on emerging patterns:

1. Target mobile platform: We made this the first decision node since each mobile platform has different models as the best performer.
2. First attempt enforcement: The second decision node is whether the usage workflow allows for iterative multi-attempt remediation or needs correctness from the very first try.
3. Self-hosting capability: This is the third decision node since data security and privacy requirements represent hard constraints that categorically eliminate certain model options. Proprietary models often use user prompts to further improve models after their initial training phase. Whatever the case is, sometimes the data is too important to be shared with external parties, making self-hostable models the only option.
4. Budget: The last decision node is the actual cost of using the model. There are two types of cost that depend on whether the model is self-hostable: 1) cost per token and 2) hardware cost to host the model. To make costs comparable across these two types, we calculated the approximate cost per token for the self-hostable models by dividing the total hosting cost of the model combined with the total tokens that are used for the experiments.

Although this study only looks at four LLMs, we chose our models to be as generalizable as possible. In terms of size (small, large), budget, type (proprietary, open source), and self-hosting capability, the resulting framework is applicable to models with comparable features. We simulated four typical mobile development scenarios to show practical applicability: Startup/Prototype Development: rapid iteration with limited resources; Production/Enterprise Development: production systems that must meet quality standards; and Self-Hosted/Privacy-Critical: requirements for regulatory compliance.

3 RESULTS

3.1 RQ1 and RQ2: TDP effectiveness for android and iOS

Test-driven prompting consistently improves performance across benchmarks and models. Twelve out of 16 model-benchmark combinations outperformed baseline accuracy, while the other four matched baseline performance, as seen in Figure 2. GPT4o on HumanEval Swift with TDP had an accuracy of 84%, while GPT4omini on MBPP Java, baseline, had an accuracy of 62%. We statistically tested 4,352 evaluations to determine TDP's generalizability across mobile development languages. TDP increased accuracy by 2.22% on average (95% CI: 1.22–3.23%). The normal distribution was confirmed by a paired t-test, which also showed a practical effect size ($d = 0.3974$) and high statistical significance ($p < 0.001$). Every model underperformed HumanEval on MBPP, suggesting that the benchmark is more complex. TDP demonstrated greater efficacy on complex problems, as evidenced by its notable gains on MBPP (+3.42%) in comparison to HumanEval (+1.03%). For remediation multi-attempt performance, the pattern is comparable but less obvious. Out of 16 model-benchmark combinations, TDP performs better than baseline in 11 of them, similarly in 2, and worse in 3. Performance varies from 64% (Qwen 2.5 14B Coder on MBPP Java, baseline) to 90% (GPT4o on HumanEval Swift, baseline). TDP results in a statistically significant increase in average accuracy of 1.98% (95% CI: 0.91–3.05%, $p < 0.0012$), even though the effect size is small ($d = 0.2911$). Nevertheless,

TDP's relative advantage is diminished by iterative error feedback, which also reduces baseline prompting effectiveness. Language-specific performance variability varies significantly: baseline prompting produces an overall accuracy range of 62%–92.7%, whereas Python-based benchmarks (such as HumanEvalPlus) display more constrained ranges (86.90%–91.30%). This implies that LLMs are less receptive to TDP engineering in mobile contexts and handle Java and Swift less skillfully than Python. Given that these benchmarks evaluate basic programming abilities, LLMs probably have a harder time with intricate, platform-specific mobile development, which calls for mobile developers to carefully examine LLM-generated code.

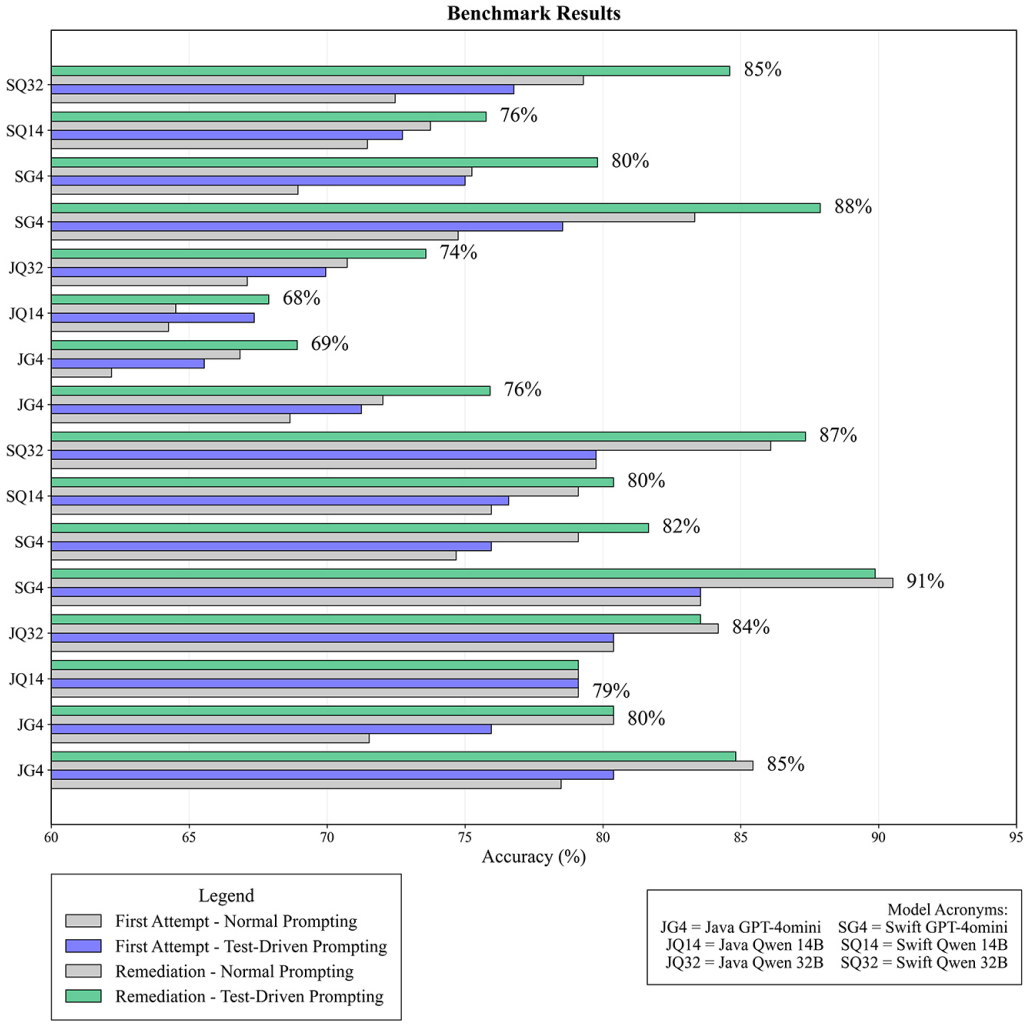

Fig. 2. Comparison of TDP accuracies and baseline prompt accuracies across benchmarks and models

Note: Accuracies are shown on a scale of 60 to 95 to emphasize the relative differences between each experiment.

3.2 RQ3: Android and iOS detailed analysis

Our objective is to offer a framework for choosing models in mobile development. To find platform-specific traits and create a suitable decision tree, we independently examined the code for iOS and Android. The average MBPP and HumanEval accuracies for each model are shown in Table 2 for Android and Table 3 for iOS, which contrasts remediation multi-attempt accuracies (R.Acc) with first-attempt accuracies (Acc @1).

Across all models, TDP continuously performs better than baseline prompting for Android (positive Delta values). GPT4o performs best in remediation evaluations (78.72% baseline, 80.36% TDP) and first-attempt evaluations (73.56% baseline, 75.81% TDP). Qwen 2.5 Coder 14B performs the worst in remediation (71.81% baseline, 73.49% TDP), while GPT4omini has the lowest first-attempt accuracy (66.85% baseline, 70.74% TDP). Across all models and evaluation types, iOS exhibits comparable trends with consistently higher accuracies. This might be because Swift is more common in LLM training data than Java for Android. The best performer is still GPT4o (remedial: 86.92% baseline, 88.87% TDP; first attempt: 79.14% baseline, 81.04% TDP). The models with the lowest performance range from Qwen 2.5 Coder 14B for TDP first attempt and all remediation evaluations (76.42% baseline, 78.07% TDP) to GPT4omini for baseline first attempt (71.81%). On both platforms, four recurring patterns are evident: (1) TDP continuously outperforms baseline prompting, (2) GPT4o performs better across the board, (3) Qwen 2.5 Coder 32B comes in second overall, and (4) GPT4omini performs worse in first-attempt evaluation but outperforms Qwen 2.5 Coder 14B in remediation.

Table 2. Android averaged accuracy

	Base Ac@1	TDP Ac@1	Delta	Base R.Ac	TDP R.Ac	Delta
GPT4o	73.56%	75.81%	+2.24	78.72%	80.36%	+1.63
GPT4omini	66.85%	70.74%	+3.89	73.61%	74.64%	+1.03
Qwen 32B	73.74%	75.16%	+1.42	77.45%	78.56%	+1.68
Qwen 14B	71.68%	73.23%	+1.55	71.81%	73.49%	+1.10

Notes: The accuracy for each model is the average accuracy of all benchmarks. TDP Ac@1 is the first attempt accuracy, while R.Ac is the remediation multi-attempt accuracy.

Table 3. iOS Averaged accuracy

	Base Ac@1	TDP Ac@1	Delta	Base R.Ac	TDP R.Ac	Delta
GPT4o	79.14%	81.04%	+1.89	86.92%	88.87%	+1.95
GPT4omini	71.81%	75.47%	+3.66	77.18%	80.72%	+3.54
Qwen 32B	76.11%	78.25%	+2.14	82.68%	85.97%	+1.26
Qwen 14B	73.70%	74.34%	+0.63	76.42%	78.07%	+1.64

Notes: The accuracy for each model is the average accuracy of all benchmarks. TDP Ac@1 is the first attempt accuracy, while R.Ac is the remediation multi-attempt accuracy.

4 DISCUSSION

A single decision framework shown in Figure 3 is adequate for both systems due to the comparable performance patterns of the iOS and Android platforms. Since LLM selection may be influenced by additional factors outside the purview of this study, this framework helps developers think through their unique use of cases and constraints without acting as a strict prescription. The usefulness of the framework is demonstrated by three application scenarios. GPT-4o-mini, which achieves 70.75% first-attempt TDD accuracy (74.65% with remediation) for Android at $0.74 per million tokens, is the best option for prototype development with limited funding and no first-attempt correctness requirements. This allows for quick iterations with controlled API costs. Second, at $12.5 per million tokens, GPT-4o Multi-Attempt offers 80.68% remediated accuracy (73.74% first-attempt) for production enterprise applications that prioritize maximum quality. Meanwhile, organizations that need data security can use managed GPT-4o

API services or self-host Qwen 2.5 Coder 32B for $5/hour. Finally, Qwen 32B Multi-Attempt (73.74% first-attempt, 78.88% remediated accuracy, $5/hour) or Qwen 14B Multi-Attempt (71.68% first-attempt, 73.50% remediated accuracy, $1.8/hour) can be self-hosted by organizations with strict data privacy requirements, like financial and healthcare institutions, providing a 64% cost reduction while maintaining compliance.

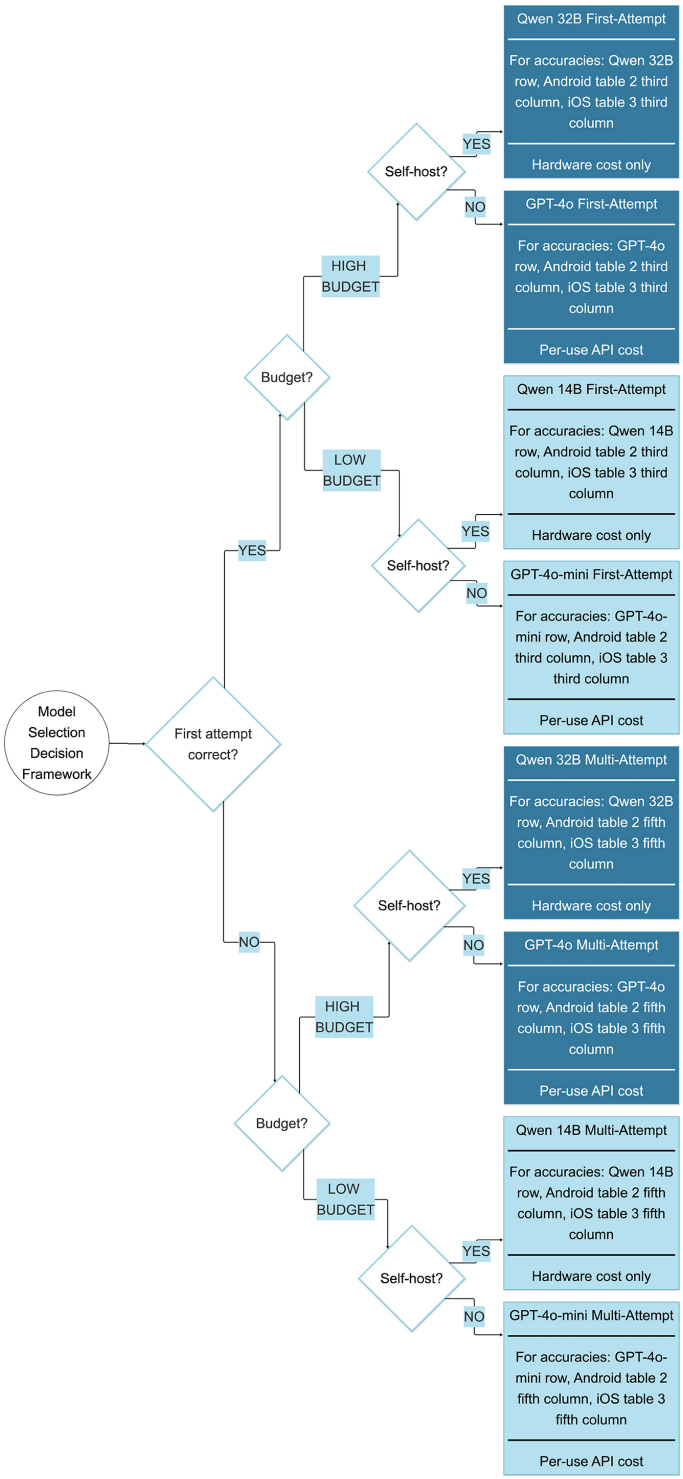

Fig. 3. Model selection framework considering correctness, budget, and self-hosting

Note: Accuracy from Tables 2 (Android) and 3 (iOS): column 3 (first-attempt), column 5 (multi-attempt with remediation).

Since the models are selected in a factorial 2×2 design that considers model size (large and small) and model types (proprietary and Open Source) which are one of the most common considerations for developers, the model decision framework would generalize to scenarios beyond the selected models in this study. Our consideration for generalizability is important considering the fast and rapid pace of LLM development in industry and academia, as every year there seem to be better and more efficient models [31].

5 CONCLUSION

The prompt engineering method known as TDP was primarily studied in Python. This is the first time that TDP has been empirically studied in mobile development with Java and Swift. We performed 8,704 evaluations across 544 programming tasks (HumanEval, MBPP) using two prompting strategies (base, test-driven) and four LLMs (GPT-4o, GPT-4o-mini, Qwen 14B, Qwen 32B). The average accuracy increase of +2.22 pp over baseline was statistically significant for TDP (95% CI [1.22–3.23 pp], $p < 0.001$, $d = 0.3974$). Regardless of model size or type (proprietary vs. open-source), our analysis shows that LLMs perform significantly worse in mobile development (66.85%–88.87%) than Python-based code generation (86.90%–91.30%). In mobile programming languages, LLMs also exhibit decreased responsiveness to TDP. We offer useful guidance for choosing use cases (max accuracy vs. budget vs. self-hosted) and platform-specific suggestions. In conclusion, our study indicates that TDP is a reliable and practically sound prompt engineering method for creating mobile applications. Because it reuses test cases that are already required in professional software engineering practices, its main benefits are its low implementation overhead and seamless integration with current development workflows. As a result, we advise both researchers and practitioners to think of TDP as a standard part of LLM-assisted mobile development approaches. Since this study only looked at mobile native programming languages, future research should look at cross-platform frameworks such as React Native and Flutter. Furthermore, the scope of this study was restricted to fundamental principles of mobile development, specifically those requiring resource optimization, platform-specific API integration, and strict compilation requirements.

6 DECLARATION OF GENERATIVE AI AND AI-ASSISTED TECHNOLOGIES IN THE WRITING PROCESS

During the preparation of this work, the author(s) used Claude 4.5 Sonnet in order to enhance grammatical accuracy, refine the writing style, and adjust the tone to align with academic conventions, while also developing the boilerplate code for generating publication-quality plot figures. After using this tool/service, the author(s) reviewed and edited the content as needed and take(s) full responsibility for the content of the publication.

7 REFERENCES

[1] Z. Ságodi, I. Siket, and R. Ferenc, "Methodology for code synthesis evaluation of LLMs presented by a case study of ChatGPT and Copilot," *IEEE Access*, vol. 12, pp. 72303–72316, 2024. https://doi.org/10.1109/ACCESS.2024.3403858

[2] X. Hou *et al.*, "Large language models for software engineering: A systematic literature review," *ACM Trans. Softw. Eng. Methodol.*, vol. 33, no. 8, pp. 1–79, 2024. https://doi.org/10.1145/3695988

[3] U. K. Durrani, M. Akpinar, H. Bektas, and M. Saleh, "Impact of artificial intelligence on software engineering phases and activities (2013–2024): A quantitative analysis using zero-truncated Poisson model," *IEEE Access*, vol. 13, pp. 95535–95547, 2025. https://doi.org/10.1109/ACCESS.2025.3574462

[4] L. Huang *et al.*, "A survey on hallucination in large language models: Principles, taxonomy, challenges, and open questions," *ACM Transactions on Information Systems*, vol. 43, no. 2, pp. 1–55, 2025. https://doi.org/10.1145/3703155

[5] I. Augenstein *et al.*, "Factuality challenges in the era of large language models and opportunities for fact-checking," *Nature Machine Intelligence*, vol. 6, no. 8, pp. 852–863, 2024. https://doi.org/10.1038/s42256-024-00881-z

[6] N. Gruver, M. Finzi, S. Qiu, and A. G. Wilson, "Large language models are zero-shot time series forecasters," in *37th Conference on Neural Information Processing Systems (NeurIPS 2023)*, vol. 36, 2023, pp. 19622–19635. Available: https://proceedings.neurips.cc/paper_files/paper/2023/file/3eb7ca52e8207697361b2c0fb3926511-Paper-Conference.pdf

[7] J. Yang *et al.*, "Harnessing the power of LLMs in practice: A survey on ChatGPT and beyond," *ACM Trans. Knowl. Discov. Data*, vol. 18, no. 6, pp. 1–32, 2024. https://doi.org/10.1145/3649506

[8] F. Tambon, A. Moradi-Dakhel, A. Nikanjam, F. Khomh, M. Desmarais, and G. Antoniol, "Bugs in large language models generated code: An empirical study," *Empir. Softw. Eng.*, vol. 30, 2025. https://doi.org/10.1007/s10664-025-10614-4

[9] J. Liu, C. S. Xia, Y. Wang, and L. Zhang, "Is your code generated by ChatGPT really correct? Rigorous evaluation of large language models for code generation," in *37th Conference on Neural Information Processing Systems (NeurIPS 2023)*, 2023, pp. 1–15. Available: https://proceedings.neurips.cc/paper_files/paper/2023/file/43e9d647ccd3e4b7b5baab53f0368686-Paper-Conference.pdf

[10] X. Xu *et al.*, "Distinguishing LLM-generated from human-written code by contrastive learning," *ACM Trans. Softw. Eng. Methodol.*, vol. 34, no. 4, pp. 1–31, 2025. https://doi.org/10.1145/3705300

[11] I. Atoum *et al.*, "Challenges of software requirements quality assurance and validation: A systematic literature review," *IEEE Access*, vol. 9, pp. 137613–137634, 2021. https://doi.org/10.1109/ACCESS.2021.3117989

[12] S. K. Pradhan, A. Kumar, and V. Kumar, "Modeling reliability-driven software release strategy considering testing effort with fault detection and correction processes: A control theoretic approach," *Int. J. Reliab. Qual. Saf. Eng.*, vol. 32, no. 2, 2025. https://doi.org/10.1142/S0218539324400023

[13] J. Yi, J. Kim, and Y. K. Oh, "Uncovering the quality factors driving the success of mobile payment apps," *J. Retailing Consum. Serv.*, vol. 77, p. 103641, 2024. https://doi.org/10.1016/j.jretconser.2023.103641

[14] L. Alwakeel, K. Lano, and H. Alfraihi, "AppCraft: Model-driven development framework for mobile applications," *IEEE Access*, vol. 13, pp. 23658–23699, 2025. https://doi.org/10.1109/ACCESS.2025.3536321

[15] O. Haggag, J. Grundy, M. Abdelrazek, and S. Haggag, "A large scale analysis of mHealth app user reviews," *Empir. Softw. Eng.*, vol. 27, no. 7, p. 196, 2022. https://doi.org/10.1007/s10664-022-10222-6

[16] B. Papis, K. Grochowski, K. Subzda, and K. Sijko, "Experimental evaluation of test-driven development with interns working on a real industrial project," *IEEE Trans. Softw. Eng.*, vol. 48, no. 5, pp. 1644–1664, 2022. https://doi.org/10.1109/TSE.2020.3027522

[17] M. Marabesi, A. García-Holgado, and F. J. García-Peñalvo, "Exploring the connection between the TDD practice and test smells—A systematic literature review," *Computers*, vol. 13, no. 3, p. 79, 2024. https://doi.org/10.3390/computers13030079

[18] S. Piya and A. Sullivan, "LLM4TDD: Best practices for test driven development using large language models," in *Proceedings of the 1st International Workshop on Large Language Models for Code (LLM4Code '24)*, 2024, pp. 14–21. https://doi.org/10.1145/3643795.3648382

[19] S. Fakhoury, A. Naik, G. Sakkas, S. Chakraborty, and S. K. Lahiri, "LLM-based test-driven interactive code generation: User study and empirical evaluation," *IEEE Transactions on Software Engineering*, vol. 50, no. 9, pp. 2254–2268, 2024. https://doi.org/10.1109/TSE.2024.3428972

[20] N. S. Mathews and M. Nagappan, "Test-driven development and LLM-based code generation," in *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering (ASE '24)*, 2024, pp. 1583–1594. https://doi.org/10.1145/3691620.3695527

[21] J. Liu *et al.*, "LLM4TDG: Test-driven generation of large language models based on enhanced constraint reasoning," *Cybersecurity*, vol. 8, 2025. https://doi.org/10.1186/s42400-024-00335-4

[22] F. Cassano *et al.*, "MultiPL-E: A scalable and polyglot approach to benchmarking neural code generation," *IEEE Transactions on Software Engineering*, vol. 49, no. 7, pp. 3675–3691, 2023. https://doi.org/10.1109/TSE.2023.3267446

[23] J. Jiang, F. Wang, J. Shen, S. Kim, and S. Kim, "A survey on large language models for code generation," *ACM Trans. Softw. Eng. Methodol.*, vol. 35, no. 2, pp. 1–72, 2025. https://doi.org/10.1145/3747588

[24] M.-F. Wong, S. Guo, C.-N. Hang, S.-W. Ho, and C.-W. Tan, "Natural language generation and understanding of big code for AI-assisted programming: A review," *Entropy*, vol. 25, no. 6, p. 888, 2023. https://doi.org/10.3390/e25060888

[25] X. Jiang *et al.*, "Self-planning code generation with large language models," *ACM Trans. Softw. Eng. Methodol.*, vol. 33, no. 7, pp. 1–30, 2024. https://doi.org/10.1145/3672456

[26] J. Jiang, F. Wang, J. Shen, S. Kim, and S. Kim, "A survey on large language models for code generation," *ACM Trans. Softw. Eng. Methodol.*, vol. 35, no. 2, pp. 1–72, 2025. https://doi.org/10.1145/3747588

[27] A. Biørn-Hansen *et al.*, "An empirical investigation of performance overhead in cross-platform mobile development frameworks," *Empir. Softw. Eng.*, vol. 25, pp. 2997–3040, 2020. https://doi.org/10.1007/s10664-020-09827-6

[28] F. Fan *et al.*, "An empirical study on common sense-violating bugs in mobile apps," *ACM Trans. Softw. Eng. Methodol.*, vol. 34, no. 6, pp. 1–26, 2025. https://doi.org/10.1145/3709356

[29] A. Ali, Y. Xia, Q. Umer, and M. Osman, "BERT based severity prediction of bug reports for the maintenance of mobile applications," *J. Syst. Softw.*, vol. 208, p. 111898, 2024. https://doi.org/10.1016/j.jss.2023.111898

[30] T. Su *et al.*, "Fully automated functional fuzzing of Android apps for detecting non-crashing logic bugs," in *Proc. ACM Program. Lang.*, 2021. https://doi.org/10.1145/3485533

[31] H. Naveed *et al.*, "A Comprehensive Overview of Large Language Models," *ACM Trans. Intell. Syst. Technol.*, vol. 16, no. 5, pp. 1–72, 2025. https://doi.org/10.1145/3744746

8 AUTHORS

Muhammad Rizqullah is a graduate student in the Department of Computer Science at King Abdulaziz University. He received his Bachelor's degree in Informatics from Telkom University, Indonesia, in 2019. He worked full-time as a Software Engineer from 2020 to 2023 at various companies, most notably at Grab, a ride-hailing tech company in Singapore. His research topics mainly include Software Engineering, Empirical Software Engineering, and Artificial Intelligence (E-mail: mrizqullah@stu.kau.edu.sa).

Emad Albassam is an Associate Professor in the Department of Computer Science at King Abdulaziz University. He received his BSc degree in computer science from King Abdulaziz University, and MSc and PhD degrees in software engineering and information technology with a concentration in software engineering from George Mason University, Fairfax, Virginia. He served as the Vice Dean for Applications at the Deanship of Information Technology. He currently serves as the Director of the Strategic Planning unit at the Faculty of Computing and Information Technology in King Abdulaziz University.