Leveraging Large Language Models for Generating Integration Test Code

Duy Hua, Adam Noack, Jenna Coffman, Anastacia MacAllister, Rey Nicolas

General Atomics

Poway, CA

duy.hua@ga-asi.com, adam.noack@ga-asi.com, jenna.coffman@ga-asi.com, anastacia.macallister@ga-asi.com, rey.nicolas@ga-asi.com

ABSTRACT

The exact amount the Department of Defense (DoD) spends on software sustainment each year isn't known, but a conservative estimate based on past budgets puts the figure close to $10 billion. Writing proper integration tests can significantly reduce these sustainment costs by reducing the number of software bugs while increasing software reliability and usability. However, writing proper integration tests is time consuming and tedious. Libraries already make the writing and running of integration tests easy, but it's still a human that must manually comb over and understand the codebase well enough to write the test logic.

Recent advancements in the AI/ML space show that large language models (LLMs) have the potential to write high quality test code, but there have been relatively few successful real-world efforts to automate the writing of integration tests with LLMs mentioned in literature.

Our study addresses a critical gap by applying an LLM-RAG pipeline specifically to integration test scenarios within an application-specific, GUI-based Python testing framework. Our small-scale real-world evaluation demonstrates a sixfold reduction in user task completion time for generating Python integration test code compared to traditional manual methods with 96% of the generated code snippets containing zero errors. We also show how the size of the knowledge base in relation to the codebase affects the quality of the downstream code generation. This work highlights practical considerations for integrating LLM-RAG solutions into existing software development workflows, addressing common challenges such as building an effective knowledge base. Ultimately, this research offers valuable guidance to the community, illustrating how to leverage advanced AI technology to accelerate software delivery, enhance reliability, and enable engineering teams to refocus on innovation and strategic tasks.

ABOUT THE AUTHORS

Duy Hua is a Software Engineer at General Atomics – ASI. At GA-ASI, Duy develops advanced HMI test automation frameworks involving object recognition and OCR. He is also building a pipeline for automated test script generation utilizing large language models (LLMs) and SBERT. His work focuses on enhancing the efficiency and intelligence of software testing and validation processes. He received his B.S. in Computer Science and M.S. in Data Analytics from Western Governors University.

Adam Noack Adam Noack is a Machine Learning Engineer at General Atomics – ASI. He received his B.A. in computer science from St. John's University in Minnesota, and his M.S. in computer science from the University of Oregon. While at UO, his research mainly focused on the interpretability and robustness of deep neural network architectures. Before working at GA, Adam worked as a machine learning research intern at several locations including Sandia National Labs, Visa, and Meta. At GA-ASI, he has, among other things, developed reinforcement learning (RL) agents that can manage their radar cross sections, measured the robustness of RL agents to adversarial attacks, and helped build a framework for testing and training RL agents in AFSIM.

Jenna Coffman is a Software Engineer at General Atomics - ASI. She received her B.S in Computer Science from San Diego State University and is currently pursuing an MBA with a specialization in Project Management from California State University, Chico. In her role, she develops automated test scripts using object recognition and OCR

technologies. Her professional focus is on expanding the use and robustness of automated testing solutions, with the goal of enhancing efficiency, reliability, and scalability in testing processes.

Dr. Anastacia MacAllister is an AI/ML Solutions Architect for General Atomics Aeronautical Systems. Her work focuses on prototyping novel machine learning algorithms, developing machine learning algorithms using sparse, heterogenous or imbalanced data sets, and exploratory data analytics. Throughout her career she has provided key contributions in a number of interdisciplinary areas such as prognostic health management, human performance augmentation, advanced sensing, and artificial intelligence aids for future warfare strategies. This work has received several awards and commendations from internal technical review bodies. Dr. MacAllister has published over two dozen peer reviewed technical papers, conference proceedings, and journal articles across an array of emerging technology concepts. She also serves on the organizing committee of the Interservice/Industry Training, Simulation & Education Conference (I/ITSEC) and contributes to the SAE G-34 committee on developing verification and validation standards for artificially intelligent aerospace systems. Dr. MacAllister holds a bachelor's degree from Iowa State University (ISU) in Mechanical Engineering. She also holds a Master's and PhD from ISU in Mechanical Engineering and Human-Computer Interaction.

Rey Nicolas is the Sr. Director of Software, Autonomy and AI Solutions and leads General Atomics Aeronautical System's Autonomy, AI, and Mission Software teams developing next generation Autonomous Command and Control (C2) systems, Autonomous Processing, Exploitation, Dissemination (PED) systems, and AI edge processing for flight and sensor autonomy, air-to-air combat, and air-to ground Intelligence, Surveillance and Reconnaissance (ISR) missions. Rey is also an Executive Committee Leader for SAE Standard Works that is developing AI in Aviation Safety Standards. Previously, Rey worked for Intel Corporation, Google leading AI R&D and product development for multiple product lines in the smart and connected home, autonomous driving, and healthcare application space. Rey holds a Bachelor of Science degree in Mechanical Engineering from University of California San Diego, Master's of Science in Computer Science from the University of Illinois, and MBA from San Diego State University.

Leveraging Large Language Models for Generating Integration Test Code

Duy Hua, Adam Noack, Jenna Coffman, Anastacia MacAllister, Rey Nicolas

General Atomics

Poway, CA

duy.hua@ga-asi.com, adam.noack@ga-asi.com, jenna.coffman@ga-asi.com, anastacia.macallister@ga-asi.com, rey.nicolas@ga-asi.com

INTRODUCTION

Software is integral to modern defense systems, underpinning everything from mission-planning tools aboard aircraft carriers to logistics management for joint operations. However, sustaining these critical software systems is costly, with the Department of Defense (DoD) spending billions annually on maintenance. A significant portion of these expenses arises from labor-intensive activities required to identify, diagnose, and fix software defects after deployment. Effective integration testing can dramatically reduce these sustainment costs by detecting problems *before* deployment. Yet, the manual creation of integration tests is resource-intensive, requiring deep domain knowledge and continuous synchronization with evolving software.

Recent advancements in large language models (LLMs) present a promising solution. These AI models, trained extensively on code and natural language data, have demonstrated proficiency in generating code and writing unit tests. Despite this progress, their practical adoption for integration testing remains limited. Challenges include the large context required for integration scenarios that exceed typical LLM capacities, inadequate domain-specific knowledge within generic models, and uncertainty about how extensively knowledge bases must be curated to ensure effective code generation.

This work addresses these challenges with two key contributions:

Firstly, we show an LLM retrieval augmented generation (RAG) pipeline for generating Python integration test code from English test procedure documents. In our small-scale human evaluation setting, this LLM-RAG pipeline helped write integration test scripts six times faster than fully manual methods, achieving 96% accuracy of generated code snippets as measured by a human expert. This pipeline leveraged a curated knowledge base consisting of paired English test procedure steps and their corresponding code snippets from our integration testing framework. We hope that our example will inspire others to speed up their own integration test writing.

Secondly, we present an analysis of knowledge base size versus code generation quality**.** Building upon previous LLM code generation studies, we systematically vary the size of our knowledge base to understand its effect on the downstream quality of generated integration test code. Our results reveal practical insights regarding minimum viable coverage of the testing framework, i.e. the percentage of the testing framework needed to be described by the knowledge base to get good downstream code generation performance. We show that there are diminishing returns to increasing knowledge base size. Furthermore, we demonstrate a simple application of k-means clustering to find relevant examples to build more optimal knowledge bases. Oftentimes in real world scenarios, curating the knowledge base is difficult, so we hope that these results will help others efficiently build effective knowledge bases of their own.

This research provides practical guidance to defense programs aiming to streamline integration test development without compromising reliability. By clearly quantifying productivity gains and offering insights into optimal knowledge base construction, we demonstrate how LLMs can be effectively integrated into existing software workflows.

BACKGROUND

Importance of Integration-Test Automation for DoD Sustainment

The US DoD manages a vast and diverse software portfolio, spanning avionics, command and control systems, logistics, etc. These systems must reliably interoperate in mission-critical environments where lives are on the line. Unfortunately, integration issues between software components often manifest post-deployment, leading to downtime in potentially critical situations. Integration tests can reduce the probability that these issues occur after deployment because they guarantee that interactions across multiple software components, services, or external interfaces work correctly.

Limitations of Traditional Automated Test Generation

Several useful tools exist for automating the writing of software test code. EvoSuite and Nessie are purpose-built to generate ready-to-run unit-test code. KLEE and SAGE produce test inputs for any entry point one chooses; one can wrap these inputs in a harness to make unit tests or high-level tests such as integration tests. To this day, these tools can outperform LLMs in generating test code in specific use cases and settings (Yang, et al., 2024). Tools such as OpenAPI Generator can turn an OpenAPI JSON ⁄ YAML document into Python (and many other) client SDKs or server stubs. Despite the usefulness of these traditional tools, they do not excel at taking in unstructured natural language descriptions and producing test code, which is the task we wish to perform.

Advances and Limits in LLM-Based Code and Unit Test Generation

Over the last several years, LLMs have shown remarkable progress in general purpose coding tasks (Wang et al., 2021) (Li, et al., 2022) (Chen, et al., 2023) (Chowdhery, et al., 2022) (Rozière, et al., 2023) (Mistral AI, 2024) (Anthropic, 2024). LLMs can now outperform some of the best humans in competitive coding problems (Jaech, et al., 2024). That said, the abilities required to solve competitive programming tasks are not quite the same as those required to write high quality test code, especially integration test code.

Several works show that while off-the-shelf LLMs show promise for generating unit tests given focal test information, they often fall short in terms of coverage when prompted naively. For example, Jain et. al show that GPT-4o covers only about 35% of focal method execution paths on real Python projects, highlighting the limits that LLMs have of reasoning about program execution (Jain, et al., 2025).TestEval benchmarks sixteen LLMs on 210 LeetCode programs and reveals that targeted line/branch/path coverage remains largely unsolved (Wang, et al., 2025).

Some works gets more creative with in-context learning by providing additional context in the prompt or iteratively re-prompting off-the-shelf LLMs with feedback and show a boost in generated unit test quality/coverage as a result. For example, Panta, an iterative hybrid of static/dynamic analysis with LLM prompting, lifts line and branch coverage over LLM-only baselines (Gu, et al., 2021). SymPrompt's code-aware, multi-stage prompting yields a 5x jump in correct tests and dramatic coverage boost, doubling GPT-4 coverage over naive prompts (Ryan, et al., 2024). AID combines variant generation with differential testing, boosting bug-finding recall up by 1.8x and precision by 2.65x over prior tools (Liu, et al., 2024). TestPilot, using GPT-3.5 and error-driven re-prompting, outperforms the non-LLM tool Nessie by 19% for statement coverage and 26% for branch coverage across 1,684 JavaScript API functions (Schäfer, et al., 2023). MuTAP augments prompts with surviving code mutants, detecting 28% more faulty programs (Moradi Dakhel, et al., 2024). A study shows that GPT-3.5/4 can generate syntactically flawless Behavior Driven Development acceptance tests with few-shot prompting (Karpurapu, et al., 2024). While these approaches are promising for unit tests they don't directly apply to integration test scenarios.

Some work finetunes LLMs specifically for the task of unit test generation. (Tufano, et al., 2020), (Rao, et al., 2023), and (Alagarsamy, et al., 2024) all show that finetuning LLMs to generate unit tests given focal information can improve their ability to generate quality tests with high coverage. That said, the data and computing resources to train such specialized models are often prohibitive in real world applications.

Integration Test Code Generation Using LLMs

Compared to the number of works that use LLMs to write unit tests, there have been very few works that explore using LLMs to generate integration test code. Li et al. (2025) tested the abilities of several LLMs to generate test cases for web form interactions, which are a common integration testing scenario for web applications. In the automotive industry, AutoUAT + Test Flow converts user stories to Gherkin then executable end-to-end test scripts, with engineers finding over 90% of the scripts helpful (Ferreira, et al., 2025). While interesting, these works are not entirely relevant to integration testing DoD code.

Retrieval-Augmented Generation (RAG) for Code Synthesis
RAG approaches leverage external knowledge bases to enhance LLM-generated outputs, bridging the gap between context window constraints and practical application needs. Parvez et al. (2021) demonstrated that finetuning an encoder using a set of parallel examples of code and summaries led to high retrieval ability and good downstream code generation. Nashid et al. (Nashid, et al., 2023) targeted retrieval of a small number of in-context examples using a SRoBERTa model finetuned on pairs of code-documentation pairs from the CodeSearchNet dataset; their approach outperforms specialized fine-tuned models in assertion generation. Zhang et al. (2023) sped up the naive k-NN neural machine translation by keeping two specialized datastores instead of one to reduce lookup time. Su et al. (2024) enhanced accuracy of their LLM-RAG pipeline by evolving the knowledge base based on the queries. Du et al. (2025) showed that building a knowledge base with the control flow and data flow representations of code snippets could improve downstream test generation performance.

The Gap Addressed by Our Work
Our study addresses a critical gap by applying an LLM-RAG pipeline specifically to integration test scenarios within an application-specific, GUI-based Python testing framework. Most previous works in automated test generation have focused on unit test generation as opposed to integration test generation and have not explored the relationship between knowledge base size and test code generation quality.

METHODS

In this section, we describe our approach to automating the generation of integration-test scripts from plain-English procedures using a RAG pipeline. First, we outline our problem setting, including the limitations of manual integration testing and conventional automation methods, and introduce our testing framework (TF), which we use to write integration test scripts that can run headless (without a GUI). We then present our specific design goals, explain our RAG pipeline architecture, and detail how we constructed and tuned the pipeline.

Our Problem and Design Goals
Modern avionics and C2 systems are mosaics of loosely coupled software components that must interoperate flawlessly in a variety of configurations. To make sure that these components can interoperate correctly, engineers run test procedures in a simulated environment to check interactions that will happen after deployment. Each test procedure is typically 5-10 pages of English text that describe a linear sequence of GUI interactions (clicks, menu selections, parameter entries) and the expected inter-component effects.

Traditionally a human tester executed these procedures manually by sitting at a workstation and exercising the GUI for between 30 minutes and several *days* per test procedure and recording pass/fail results. This approach did not scale. Our release cadence demands many tests after every build and the unavoidable operator inconsistencies introduced variability into the results.

To eliminate one “human-in-the-loop” bottleneck, the TF was built. It exposes every GUI control as a Python API so that a complete integration test can be executed headless. Once an automated test script has been reviewed and vetted, it can be executed without a human in the loop as many times as needed at virtually no cost.

Although the TF reduces costs at the time of test *execution*, prior to the creation of our pipeline, *authoring* a TF script still required a test engineer to translate every English instruction into a sequence of TF API calls. This task of manually translating a single English test procedure document into Python TF code could consume 30–60 minutes for an experienced engineer and 1-2 days for a novice. It took this long even though the TF is readable and maintainable with a cyclomatic complexity of less than 2 and a maintainability index in the 70s and most test procedures map somewhat cleanly onto a combination of TF commands. The lengthy time required to write new tests might be partially due to the TF’s size, which has over 100k lines of Python code spread over 300 modules.

We wished to speed up the time required to write a TF test script. However, conventional program-synthesis tools proved inadequate for the task of translating English instructions into TF commands: they require rigid grammar and cannot generalize to the domain-specific vocabulary found in procedures. Furthermore, simple keyword searches across both the TF and previously written test scripts failed to speed test engineers up due to small formatting discrepancies between keywords and code. Moreover, even if somewhat relevant code was returned successfully with a simple keyword search, the code that was returned almost never had a correct combination of TF calls and would necessitate human editing. Consequently, we turned to RAG to build our text-to-code translation solution.

Our solution needed to satisfy three main goals. First, it needed to provide automatic translation, meaning that given a plain English language procedure, the system needed to emit a syntactically correct TF script that calls the appropriate TF code for each step. Second, the output needed to be human-verifiable; we assume a test engineer remains in the loop as final arbiter, thus generated code must be concise and readable. Third, the script needed to be edit-efficient, being mostly correct on the initial pass.

Pipeline Architecture

The RAG pipeline for writing our integration test code is shown below in Figure 1. The main components are the knowledge base KB containing N pairs of English test procedure steps and their corresponding Python commands that use the testing framework, $KB = \{(e_i, c_i)\}_{i=1}^{N}$, the embedding model (i.e. the embedder) $\phi\colon\ \mathcal{X} \to \mathbb{R}^d$, and the LLM. Alongside each pair in the knowledge base is the precomputed embedded test procedure step $\phi(e_i)$.

Each new test procedure $\tilde{E}$ is a sequence of M independent steps $(\tilde{e_1}, \tilde{e_2}, \dots, \widetilde{e_M})$. Our pipeline takes each step of this test procedure $\tilde{e_l}$ in, one at a time. The step is embedded $\phi(\tilde{e_l})$ and the k-nearest procedure steps from the knowledge base along with their code snippets $\mathcal{N}_k(\tilde{e_l}) = \{(e_j, c_j)\}_{j=1}^{k}$ are obtained. These reference pairs $\mathcal{N}_k(\tilde{e_l})$ along with the new test step $\tilde{e_l}$ are injected into the prompt and sent to the LLM. Given these reference examples, the LLM will generate TF Framework code for $\tilde{e_l}$. This will be repeated for each of the M steps of the test procedure $\tilde{e_1}, \tilde{e_2}, \dots, \widetilde{e_M}$ sequentially and independently.

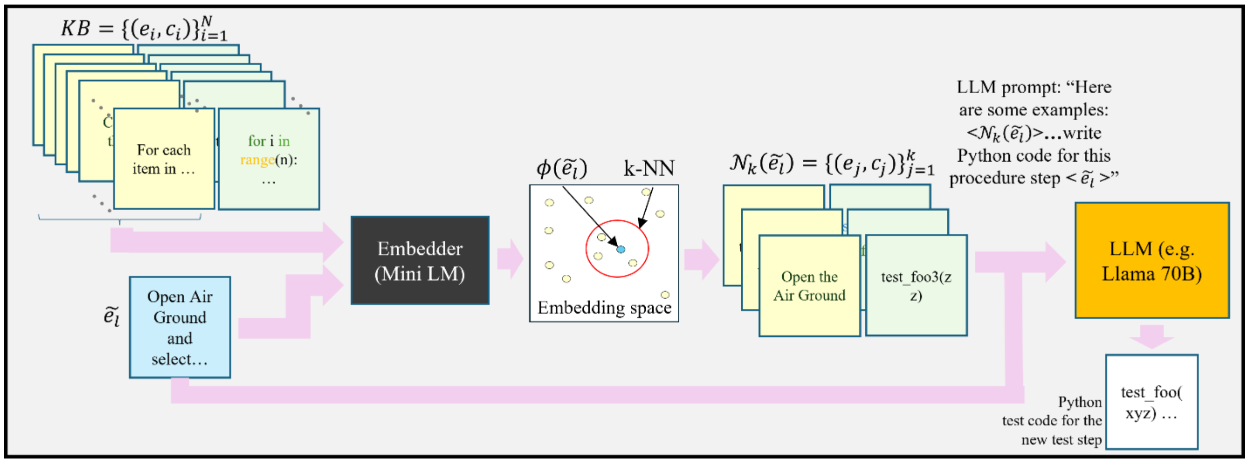

$KB = \{(e_i, c_i)\}_{i=1}^{N}$
For each item in …
for i in range(n): …
$\tilde{e_l}$
Open Air Ground and select…
Embedder (Mini LM)
$\phi(\tilde{e_l})$
k-NN
Embedding space
$\mathcal{N}_k(\tilde{e_l}) = \{(e_j, c_j)\}_{j=1}^{k}$
Open the Air Ground
test_foo3(z z)
LLM prompt: "Here are some examples: <$\mathcal{N}_k(\tilde{e_l})$>…write Python code for this procedure step <$\tilde{e_l}$>"
LLM (e.g. Llama 70B)
Python test code for the new test step
test_foo(xyz) …

Figure 1: Our RAG pipeline. The pipeline accepts a single new English test procedure and produces the corresponding TF Python code snippet.

Knowledge Base

We utilize a KB consisting of pairs of N English procedure steps and clear demonstrations of our Python TF usage. KB has 3,600 total pairs, and for many experiments, $N = 3{,}500$ with 100 pairs set aside for evaluation.

Table 1: Two dummy samples that match the format of real samples from our knowledge base. The code snippets (right column) are usually between 1 and 20 lines of code.

Test Procedure Step	TF Python Code Snippet
Move the stick back to ground idle (Detent 35)	`self.sim.set("stickAxis", 35)`

Verify that the "MWIR Iris Selection" picker button title is "LWIR". [TBL->XPO->Camera1->LWIR->LAP]	`self.assertTrue(`
	`  self.pgcs.p_cmd.ams.is_button_value_present(`
	`    button_value="LAP", group="LWIR"))`

As mentioned above, the TF Python codebase is quite large with over 100k lines of code. However, relatively few methods in the TF—just over one thousand—are *external-facing methods*. What this means is that, for a human to write a new testing script, a relatively small portion of the TF needs to be referenced directly. Therefore, external-facing method examples are prioritized in KB. Despite this fact, only about 17% of these external facing methods are referenced in the KB of size $N = 3{,}500$. However, it's clear in the results below that this low coverage value of 17% does not hinder downstream code generation performance.

Embedding Model and Retrieval from the Knowledge Base
We use the sentence-transformers/all-MiniLM-L6-v2 transformer for our embedding model ϕ from HuggingFace.com. It maps sentences and paragraphs to a 384-dimensional dense vector space. It was trained on several code related datasets such as CodeSearchNet, so it works well for our application off-the-shelf. It is ideal for clustering and semantic search tasks and is quite lightweight with only 22M parameters. In all our experiments, we use the cosine similarity metric at inference time to find similar samples from our knowledge base.

At inference time, a single input test procedure step $\widetilde{e_l}$ is embedded by the embedding model $\phi(\widetilde{e_l})$, and this vector is compared using cosine similarity to all the embeddings for test procedure steps from the knowledge base $\{\phi(e_i)\}_{i=1}^{N}$. The top-k most similar test procedure steps and their paired code snippets $\mathcal{N}_k(\widetilde{e_l}) = \{(e_j, c_j)\}_{j=1}^{k}$ are pulled from KB and injected into the LLM prompt. In some experiments, we use a scalar threshold τ to filter out pairs with cosine similarity values lower than τ.

Finetuning the Embedding Model
We tested two versions of this embedding model. For most experiments, we used the embedding model off-the-shelf with the unmodified weights from HuggingFace.com. For some experiments though, we tried finetuning the embedding model using multiple negatives contrastive loss (Henderson, et al., 2017), as shown in Equation 1 below, to better align similar English procedure steps in the embedding space.

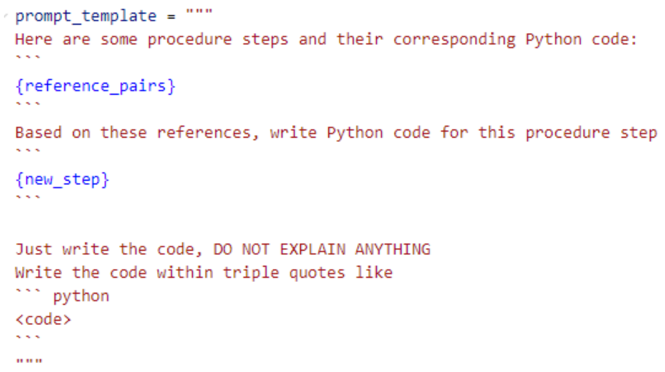
$$J(E, C, \theta) = -\frac{1}{K}\sum_{i=1}^{K}\left[S(e_i, c_i) - \log\sum_{j=1}^{K} e^{S(e_i, c_j)}\right] \qquad (1)$$

If θ is the word embeddings and learnable network weights, E is a set of English test procedure steps from KB, C is the corresponding set of code snippets in KB, and K is the number of samples in E and in C, the objective is to minimize the loss J, by maximizing the similarity between correct pairs $S(e_i, c_i)$ minimize the similarity between incorrect pairs $S(e_i, c_j)$ where $i \neq j$. The similarity function is a dot product between the embedded versions of e_i and c_i. Intuitively, what's happening is that we're forcing each English test procedure step e_i to be close to the correct Python snippet c_i in the embedding space while forcing e_i to be far from all other Python snippets $c_{j\neq i}$ in the embedding space.

We trained the embedding model with this loss function using 3,200 pairs of English and Python pairs for training data, 200 for validation and 200 for testing. With this finetuned model, we could use *English* test step queries to search across the entire TF codebase to look for Python code snippets relevant to the English query.

LLMs Used and Prompt Design and Choice for k
We use three different LLMs in our experiments: Llama 3.3 8B, Llama 3.3 70B, and Amazon's Nova Pro. None of these models was hosted on premises, rather all were hosted securely on Impact Level (IL) 4 Amazon Bedrock GovCloud servers and were queried using Bedrock API calls.

We designed the few-shot prompt template as shown in Figure 2. Note that because many modern LLM context windows are massive, e.g., Llama 3.3 has a context length of 128k tokens, it seems intuitive to include significant amounts of somewhat relevant information into the prompt of the LLM and let it sort out the details needed to produce a correct output. However, doing this can often be detrimental to performance (Yang, et al., 2024) (Nashid, et al., 2023) (Jain, et al., 2025). In-context learning with a few specific correct examples often produces better results than putting tons of data into the prompt. Another downside of large prompts is that they typically result in slower and more costly inference. For these reasons, we tested relatively small values for k, the number of examples to pass to the LLM in prompt.



```
prompt_template = """
Here are some procedure steps and their corresponding Python code:
```
{reference_pairs}
```
Based on these references, write Python code for this procedure step
```
{new_step}
~~~

Just write the code, DO NOT EXPLAIN ANYTHING
Write the code within triple quotes like
``` python
<code>
~~~
"""
```

Figure 2: Our prompt template; reference_pairs is a list of the k nearest (English step, Python TF code snippet) pairs from the knowledge base.

Evaluation Set and Metrics

We set aside 100 of the 3,600 pairs—of English steps and their correct Python code snippets—to serve as an evaluation set. We used this same evaluation set across all the experiments. We followed previous works and tracked performance across several commonly used evaluation metrics (Nashid, et al., 2023):

- Abstract syntax tree (AST) match: like an exact string match metric, but ignores slight formatting differences between the compared strings
- Longest common subsequence (LCS): the ratio of the longest common subsequence between the predicted output and the ground truth output
- Edit distance (ED): the number of character-level edit operations (add, delete, modify) needed to transform the generated output into the ground truth output
- Char-BLEU (CB) similarity: a BLEU score over character-level n-grams

In our small-scale, real-world experiment, expert human reviewers manually combed over the lines of code that were generated by the pipeline and counted the number of lines that were usable without any modification necessary. Note that generated code can often be acceptable even with a low AST, LCS, ED, and CB score.

RESULTS AND DISCUSSION

In this section we will describe the set of experiments we ran and discuss their results. For all experiments, unless otherwise specified, we use the following default parameter values. We use a knowledge base KB size of $N = 3{,}500$ samples, we use the un-finetuned embedding model ϕ, we use a threshold value τ of 0 (to keep k consistent from query to query), we use Llama 3.3 70B with the temperature set low at 0.2 to reduce output volatility, and we evaluate on our 100-sample benchmark.

The Effect of Varying k

In Table 2, we systematically vary the few-shot depth $k \in \{1, 3, 5, 7, 9\}$ to determine how additional demonstrations influence end-to-end RAG accuracy.

As can be seen, $k = 5$ or $k = 7$ produces optimal results, depending on the metric of choice. As seen in other works, there seems to be diminishing returns to increasing k. It's unclear to what extent there are diminishing returns because as k increases each additional example added to the reference set is slightly less like the query than the previously added example.

k	AST	LCS	ED	CB
1	0.29	0.689	0.702	0.687
3	0.31	0.709	0.722	0.718
5	0.32	**0.734**	**0.750**	**0.745**
7	**0.36**	0.709	0.739	0.735
9	0.3	0.71	0.727	0.726

Table 2: The effect of varying k.
```

**The Effect of Introducing a Threshold Value**

In Table 3, instead of having a fixed number of examples provided to the LLM in the prompt, we introduce a threshold value $\tau$, so that $k$ effectively becomes the *maximum* number of examples that will be provided to the LLM in the prompt, i.e., only those examples $(x_i, y_i)$ where the cosine similarity with the query is greater than $\tau$ are placed in the prompt. For each value of $\tau$, we keep $k = 7$.

Interestingly, a threshold value of 0.2 produces the best results. This is much lower than the default range of 0.6-0.8 that is typically suggested in literature.

| $\tau$ | AST | LCS | ED | CB |
|---|---|---|---|---|
| 0 | **0.36** | 0.709 | 0.739 | 0.735 |
| 0.2 | 0.33 | **0.735** | **0.755** | **0.759** |
| 0.4 | 0.34 | 0.717 | 0.743 | 0.736 |
| 0.6 | 0.33 | 0.724 | 0.747 | 0.745 |
| 0.8 | 0.35 | 0.697 | 0.719 | 0.7 |

**Table 3: The effect of introducing a threshold value in the retrieval process.**

**The Effect of Changing the LLM**

In Table 4, we vary the LLM being used and test out two values for $k$ for each LLM.

As seen, Llama 70B and Nova Pro do significantly better than the smaller Llama 8B.

| LLM | $k$ | AST | LCS | ED | CB |
|---|---|---|---|---|---|
| Llama 8B | 1 | 0.25 | 0.557 | 0.574 | 0.558 |
| Llama 8B | 5 | 0.25 | 0.61 | 0.623 | 0.608 |
| Llama 70B | 1 | 0.3 | 0.683 | 0.699 | 0.672 |
| Llama 70B | 5 | 0.33 | 0.707 | **0.741** | **0.73** |
| Nova Pro | 1 | 0.18 | 0.643 | 0.659 | 0.663 |
| Nova Pro | 5 | **0.34** | **0.723** | 0.737 | 0.724 |

**Table 4: Effect of different LLMs**

**RAG Directly on TF Code and Finetuned Embedding Model**

In Table 5, we test what happens when we don't use our English step / Python code snippet knowledge base at inference time but instead use the chunked up TF itself as the knowledge base.

The first row are the results when a similarity search between an embedded English procedure step query and embedded chunks of TF code is done. We supply the most similar $k = 5$ TF chunks to the LLM in the prompt and generate the embeddings using the unfinetuned embedding model.

We also tried finetuning the embedding model using multiple negatives contrastive loss (Henderson, et al., 2017) as mentioned above on our English step / Python code snippet knowledge base. This finetuned embedding model is therefore better at aligning English procedure steps and relevant TF code chunks in the embedding space.

| Embedder | AST | LCS | ED | CB |
|---|---|---|---|---|
| Unfinetuned | 0.02 | **0.28** | **0.34** | **0.32** |
| Finetuned | **0.04** | **0.28** | 0.33 | 0.3 |

**Table 5: Experiments with using TF snippets directly instead of pairs from $KB$.**

Given that the unfinetuned version of the embedding model was already trained on code search datasets, it was unsurprising that there was almost no difference in downstream generation performance between the unfinetuned and finetuned embedding models. However, it was surprising how poorly Llama 70B was at generating code when it was only supplied with relevant TF chunks without paired up English procedure steps.

**The Effect of Knowledge-Base Size and Knowledge Base Selection Process**

To understand the trade-off between the size of the knowledge base and downstream generation performance, we started with a new, empty knowledge base $KB^*$ and randomly (uniform) pulled 10, 20, 40, etc. samples from the original knowledge base $KB$ into $KB^*$. For each size of $KB^*$ we tested the pipeline's downstream generation performance using the default parameters mentioned in the first paragraph of this section.

We also used k-means clustering instead of random sampling to build a more optimally representative knowledge base for each size. For each $KB^*$ size $N^*$, we set the number of clusters for the k-means algorithm to $N^*$. Then we ran k-means to find $N^*$ cluster centers. For each cluster center, we chose the sample in the embedding space that was closest to the center to be the representative sample for that cluster. Then and we aggregated all these representative samples together to form $KB^*$. Figure 3 shows the performance trend across smaller knowledge base sizes (up to 640 samples) and Table 6 shows an expanded set of results.

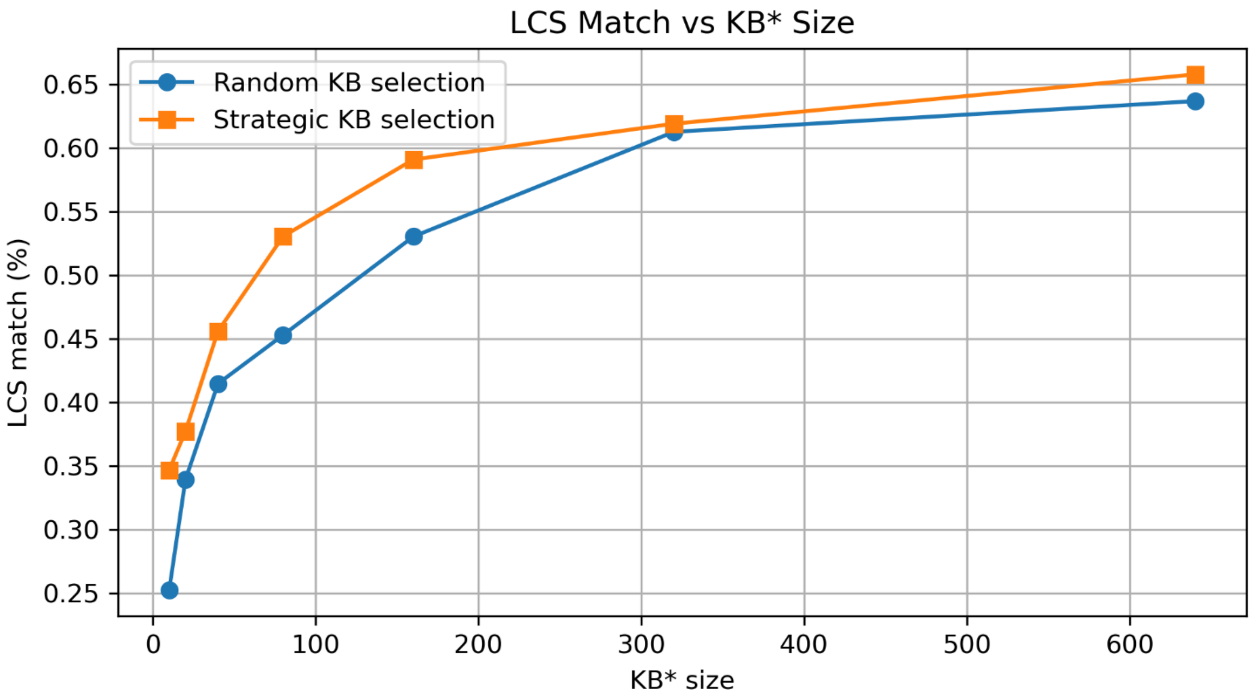


**Figure 3: The diminishing performance gains of increasing the knowledge base size. Note that for all KB sizes, the k-means "Strategic KB selection" process yields better downstream performance than the random selection process.**

**Table 6: Expanded results from Figure 2. The effect of changing the knowledge base size. Simple k-means selection of $KB^*$ reliably improves downstream generation for every size.**

| | | $KB^*$ Size | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| **$KB^*$ Selection** | **Metric** | **10** | **20** | **40** | **80** | **160** | **320** | **640** | **1280** | **2560** |
| Random | AST | 0.01 | 0.02 | 0.05 | 0.05 | 0.17 | **0.19** | 0.19 | 0.28 | 0.38 |
| k-means | AST | **0.03** | **0.04** | **0.08** | **0.13** | **0.18** | **0.19** | **0.24** | **0.30** | **0.34** |
| Random | LCS | 0.25 | 0.34 | 0.41 | 0.45 | 0.53 | 0.61 | 0.64 | 0.68 | **0.74** |
| k-means | LCS | **0.35** | **0.38** | **0.46** | **0.53** | **0.59** | **0.62** | **0.66** | **0.72** | 0.72 |

Figure 3 and Table 6 show that there are diminishing gains to increasing the size of $KB^*$. It seems that after a significant fraction of the external-facing TF methods is represented in the knowledge base, additional examples add less value. The same plot and table highlight the efficiency of k-means sampling; the performance of k-means matches that of a randomly selected knowledge base roughly *twice* its size. Therefore, investing time in choosing representative samples for the knowledge base (via k-means or a similar strategy) is worthwhile.

**Small-Scale Human Evaluation**

We applied our RAG pipeline to a representative sample of 268 novel integration test procedure steps, chosen specifically to cover a diverse range of possible cases. For this evaluation, we used the Llama 8B model with a retrieval count $k$ of 5 and a similarity threshold of 0.8, alongside the default parameters described earlier.

The RAG pipeline produced a total of 1,982 lines of Python code from these 268 procedure steps. A human reviewer then conducted a meticulous line-by-line review and made manual corrections to any inaccuracies. Remarkably, only 12 lines required editing, indicating a line accuracy rate of approximately 99%, with just 1% of lines requiring modification. These 12 line errors were spread evenly across the code for 12 different procedure steps—in other words, the RAG pipeline produced flawless Python code for 256 of the 268, or 96%, of the test procedure steps. Completing this verification and editing task took the reviewer only one hour. In comparison, we estimated that a junior engineer, using a modern development environment such as VS Code and having convenient access to existing tests, would require at least 6 hours to produce comparable results manually. Consequently, the RAG-assisted approach demonstrated nearly a sixfold improvement in productivity. We also have strong evidence that this substantial timesaving extends effectively to larger-scale production applications.

These results are particularly impressive given that the code snippets in our knowledge base covered only 17% of the external-facing methods in the TF. We attribute this success to the TF's inherently low cyclomatic complexity and high maintainability. These characteristics likely allowed the language model to effectively infer the names of methods and parameters not explicitly included in the knowledge base by referencing structurally similar entries.

## CONCLUSION

Rapid, reliable integration-test authoring is a pressing need across defense and industrial software programs. Our study shows that a retrieval-augmented LLM pipeline can cut script-writing effort by nearly 6x while achieving a 96% first-pass acceptance rate. Furthermore, we show the relationship between knowledge base size and downstream generation performance and demonstrate in our case that an intelligent knowledge base creation process can be 2x more efficient than a random knowledge base curation process.

Our work demonstrates that with the right LLM, embedding model, and knowledge base, large-scale automation of integration test code generation is already feasible. The approach delivers tangible cost savings and shortens release cycles, giving engineering teams headroom to expand test coverage instead of fighting tooling overhead.

## REFERENCES

Alagarsamy, S., Tantithamthavorn, C., & Aleti, A. (2024). A3Test: Assertion-augmented automated test case generation. *Information and Software Technology, 176*, 107565.

Chen, B., Zhang, F., Nguyen, A., Zan, D., Lin, Z., Lou, J.-G., & Chen, W. (2023). CodeT: Code generation with generated tests. *International Conference on Learning Representations.*

Chowdhery, A., Narang, S., Devlin, J., & Dean, J. (2022). *PaLM: Scaling language modeling with Pathways.* The Journal of Machine Learning Research.

Claude 3.5 Sonnet. (2024, 6 21). Anthropic. Retrieved from https://www.anthropic.com/news/claude-3-5-sonnet

Codestral: Empowering developers (2024, 5 29). Mistral AI. Retrieved from https://mistral.ai/news/codestral

Du, K., Chen, J., Rui, R., Chai, H., Fu, L., Xia, W., & Zhang, W. (2025). *CodeGRAG: Bridging the gap between natural language and programming language via graphical retrieval augmented generation.* arXiv. Retrieved from https://arxiv.org/abs/2405.02355

Ferreira, M., Viegas, L., Faria, J., & Lima, B. (2025, 4). Acceptance test generation with large language models. *arXiv.* doi:10.48550/arXiv.2504.07244

Gu, J., & Chen, Z. (2021). *Multimodal representation for neural code search.* IEEE. Retrieved from https://arxiv.org/abs/2107.00992

Henderson, M., Al-Rfou, R., Strope, B., Sung, Y., Lukács, L., Guo, R., . . . Kurzweil, R. (2017). *Efficient Natural Language Response Suggestion for Smart Reply.* arXiv. Retrieved from https://arxiv.org/abs/1705.00652

Jaech, A., Kalai, A., Lerer, A., Richardson, A., El-Kishky, A., Low, A., & Metz, L. (2024, December). *OpenAI o1 system card.* Retrieved from https://openai.com/index/openai-o1-system-card/

Jain, K., Synnaeve, G., & Rozière, B. (2025). TestGenEval: A real-world unit test generation and test completion benchmark. *International Conference on Learning Representations.*

Karpurapu, S., Myneni, S., Nettur, U., Gajja, L., Burke, D., Stiehm, T., & Payne, J. (2024). Comprehensive evaluation and insights into the use of large language models in the automation of behavior-driven development acceptance test formulation. *IEEE Access, 12*, 1–14.

Li, T., Cui, C., Towey, D., Huang, R., & Ma, L. (2025). Large Language Models for Automated Web-Form-Test Generation: An Empirical Study. *ACM Transactions on Software Engineering and Methodology*.

Li, Y., Choi, D., Chung, J., Kushman, N., Schrittwieser, J., Leblond, R., & Vinyals, O. (2022). Competition-level code generation with AlphaCode. *Science, 378*(6624), 1092–1097.

Liu, K., Liu, Y., & Huang, G. (2024, 4 16). *LLM-Powered Test Case Generation for Detecting Tricky Bugs.* arXiv. Retrieved from https://doi.org/10.48550/arXiv.2404.10304

Moradi Dakhel, A., Nikanjam, A., Majdinasab, V., Khomh, F., & Desmarais, M. (2024). Effective test generation using pre-trained large language models and mutation testing. *Information and Software Technology, 171*, 107468.

Nashid, N., Sintaha, M., & Mesbah, A. (2023). Retrieval-based prompt selection for code-related few-shot learning. *45th International Conference on Software Engineering (ICSE 2023)*, (pp. 2450–2462). doi:10.1109/ICSE48619.2023.00205

Parvez, M., Ahmad, W., Chakraborty, S., Ray, B., & Chang, K.-W. (2021). Retrieval augmented code generation and summarization. *Empirical Methods in Natural Language Processing*, (pp. 2719–2734).

Rao, N., Jain, K., Alon, U., Le Goues, C., & Hellendoorn, V. (2023). CAT-LM: Training language models on aligned code and tests. *Automated Software Engineering*, (pp. 409–420).

Rozière, B., Gehring, J., Gloeckle, F., Sootla, S., Gat, I., Tan, X., & Synnaeve, G. (2023). *Code Llama: Open foundation models for code.* arXiv. Retrieved from https://arxiv.org/abs/2308.12950

Ryan, G., Jain, S., Shang, M., Wang, S., Ma, X., Ramanathan, M., & Ray, B. (2024). Code-aware prompting: A study of coverage-guided test generation in a regression setting using LLMs. *Proceedings of the ACM on Software Engineering, 1*(FSE), Article 43.

Schäfer, M., Nadi, S., Eghbali, A., & Tip, F. (2023). An empirical evaluation of using large language models for automated unit test generation. *IEEE Transactions on Software Engineering, 50*(1), 85–105.

Su, H., Jiang, S., Lai, Y., Wu, H., Shi, B., Liu, C., . . . Yu, T. (2024). EVOR: Evolving retrieval for code generation. *Empirical Methods in Natural Language Processing.*

Tufano, M., Drain, D., Svyatkovskiy, A., Deng, S., & Sundaresan, N. (2020). *Unit test case generation with transformers and focal context (AthenaTest).* arXiv. doi:arXiv:2009.05617

Wang, Y., Wang, W., Joty, S., & Hoi, S. (2021). CodeT5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. *Empirical Methods in Natural Language Processing*, (pp. 8696–8710).

Wang, Z., Asai, A., Yu, X., Xu, F., Xie, Y., Neubig, G., & Fried, D. (2025). *CodeRAG-Bench: Can retrieval augment code generation?* North American Chapter of the Association for Computational Linguistics.

Yang, L., Yang, C., Gao, S., Wang, W., Wang, B., Zhu, Q., & Chen, J. (2024). On the evaluation of large language models in unit test generation. *Automated Software Engineering*, (pp. 1607–1619).

Zhang, X., Zhou, Y., Yang, G., & Chen, T. (2023). Syntax-aware retrieval augmented code generation. *Empirical Methods in Natural Language Processing*, 1291–1302.