Examples out of Thin Air: AI-Generated Dynamic Context to Assist Program Comprehension by Example

Toni Mattis
Hasso Plattner Institute
University of Potsdam
Potsdam, Germany
toni.mattis@hpi.uni-potsdam.de

Eva Krebs
Hasso Plattner Institute
University of Potsdam
Potsdam, Germany
eva.krebs@hpi.uni-potsdam.de

Martin C. Rinard
Massachusetts Institute of Technology
Cambridge, MA, USA
rinard@csail.mit.edu

Robert Hirschfeld
Hasso Plattner Institute
University of Potsdam
Potsdam, Germany
robert.hirschfeld@uni-potsdam.de

ABSTRACT

Programmers often benefit from the availability of concrete run-time data alongside abstract source code. However, programmers need to manually exercise the program to reach an interesting state or write code that reproducibly executes a functionality with concrete inputs to be able to observe concrete data.

This work aims to automate this process by leveraging generative AI. We present a framework and a preliminary Smalltalk-based prototype allowing programmers to obtain and run examples for the currently viewed source code section from a large language model.

Our approach demonstrates how locally hosted LLMs can be fine-tuned and used for such a task with reasonable computational effort while minimizing common problems like hallucinations and out-of-date knowledge. The framework has direct applications in example-based live programming, where it can suggest new examples, and in learning settings where novices need to know how to use certain functionality.

CCS CONCEPTS

• **Software and its engineering** → **Integrated and visual development environments**; Automatic programming; • **Computing methodologies** → **Natural language processing**.

KEYWORDS

live programming, example-based programming, generative ai, large language models, smalltalk

ACM Reference Format:
Toni Mattis, Eva Krebs, Martin C. Rinard, and Robert Hirschfeld. 2024. Examples out of Thin Air: AI-Generated Dynamic Context to Assist Program Comprehension by Example. In *Companion Proceedings of the 8th International Conference on the Art, Science, and Engineering of Programming (‹Programming› Companion '24), March 11–15, 2024, Lund, Sweden.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3660829.3660845

1 INTRODUCTION

Source code is inherently abstract, yet our understanding of abstract procedures benefits from concrete examples. Worked examples have been used in education since the emergence of written history [10] and continue to be effective today [13]. Several practices in software engineering, such as testing and technical documentation, use examples that illustrate the workings of software.

In modern programming environments, programmers have several options to understand a program by example: they can use print statements, debugging facilities to halt a running program, inspect run-time data, and step through the program to observe how data changes. Several advanced mechanisms have since been proposed that bring concrete data even closer to the source code and illustrate its effects over time, such as omniscient debugging [18], Example-centric Programming [3], Exemplars [1], bimodal example-based programming [7], and Babylonian Programming [15].

A question that concerns all of these practices is how programmers can come up with examples or steer the program into a state that serves as example to demonstrate the logic they are trying to understand.

Typically, programmers manually interact with the running program until it is in the state of interest (and drop into a debugger or live inspector), isolate parts of the program to run in a unit test, or use interactive code execution in a debugger or REPL to explore its behavior. Live examples in Babylonian Programming, Exemplars, or Example-centric Programming settings need to be manually created or "mined" from manually created test cases or program runs.

A novel opportunity to help programmers understand code by example presents itself with the recent emergence of large language models (LLMs). An LLM is trained on natural language and coding tasks – likewise "by example" – allowing us to fine-tune it to generate suitable examples in a specific domain for previously unseen code.

Automating the Example. In this work, we are proposing an approach to *automatically* generate live examples and a preliminary prototype in the Squeak/Smalltalk live programming environment. We leverage an open-source LLM to generate realistic invocations of the code of interest and execute the generated code in the live programming environment. This allows programmers to obtain a running instance of the part of the system they are currently studying. Our approach is based on the automated synthesis of *tests* but focuses on set-up and execution rather than coverage and assertions to verify behavior.

Pre-trained code LLMs have several limitations that we address in this work: (1.) They lack knowledge about Squeak/Smalltalk and cannot reliably distinguish between different Smalltalk distributions and versions. (2.) They lack specific and up-to-date knowledge about the program the programmer is trying to understand. (3.) They are trained on generic code completion or instruction following and lack task-specific schemas they can apply to generate examples.

We address these challenges by (1.) fine-tuning an LLM on the Squeak/Smalltalk image and the project the programmer currently works with. Fine-tuning uses a data augmentation strategy that attempts to put callers of methods in front of callees, thereby "teaching" the LLM to generate realistic state and input data, and (2.) presenting the LLM with relevant context surrounding the code that requires an example.

The outcome is a framework that can integrate with existing example-based live programming environments that subsequently aid with using and maintaining generated examples.

2 BACKGROUND AND RELATED WORK

First, we provide a (simplified) overview of the principles underlying (large) language models and recent developments we will exploit in this work. After that, we review some example-based programming systems to illustrate the programming tools that would benefit directly from automated example generation.

2.1 LLMs on Code

A language model is a function that, given a sequence of tokens, computes probabilities for the next token following that sequence. This stateless function alone does not lend itself to generating code, but starting with an initial sequence (prompt) and repeatedly selecting among the most probable subsequent tokens, then appending one to the current sequence for the next iteration generates a theoretically unbounded sequence (completion) as illustrated in Figure 1. Generation stops when it reaches a special end-of-sequence token or a pre-configured maximum length.

Tokens. Tokens in this context are independent of the language’s grammar. They represent the most common substrings in the training data: programming language keywords and frequent English words form a single token[1]. Less frequent words are split into smaller tokens (e.g., "Smalltalk" as `Small` + `talk`), with individual letters as a fallback. Current language models use between 30 000 and 52 000 individual tokens, including special tokens like the one representing the end of the sequence (⟨*EOS*⟩).

Generation Strategies. There are several ways to select tokens from the language model: A *greedy* strategy would repeatedly select and append the most likely token. However, this can miss better

[1]Curiously, that leads to the names of US presidents being single tokens.

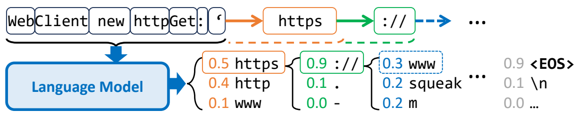


Figure 1: Generating code from a language model by repeatedly selecting a likely token and appending it to the current context.

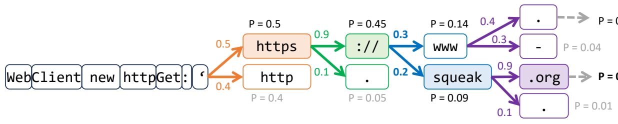


Figure 2: Search-based code generation keeps multiple examples in memory and accumulates their total probability. Only the most likely paths are continued, resulting in better completions that hide behind tokens with lower individual probability.

completions hiding behind the less likely tokens. A *search-based* strategy keeps a set of completion candidates in memory (storing their total probability). It incrementally extends each candidate with several likely tokens, discarding candidates with the lowest total probability. *Beam search* is a search-based strategy that can be implemented efficiently on GPUs, keeping k completions in memory and extending each by the top-k tokens, discarding down to k after sorting by total probability (illustrated in Figure 2 for $k = 2$). *Sampling* is a strategy where each token is picked randomly according to its associated probability.

Search is a good choice if a single attempt at generating code should be the best the model can output, while sampling allows generating diverse completions.

State-of-the-Art Models. State-of-the-art language models, such as *CodeLlama* [16], WizardCoder [12], or *StableCode* [14], are *transformers* that chain two lookup mechanisms: The first one, called *attention*, identifies for each token which other tokens are relevant, considering both the token itself and its relative position (e.g. if the token is an adjective, the attention mechanism might look for a `not` token before as that would negate its meaning). The updated representation of this "combined" context is fed into a neural network that looks up the probabilities of the following tokens based on their appearance in the training data. Several of such transformer blocks collectively vote for the next token, each one seeing the previous output, i.e., each new attention layer can now further combine previously combined contexts to learn higher-level abstractions and refine the previous prediction.

The operations mostly rely on matrix multiplication, with matrix elements making up the *parameters* of the model. State-of-the-art language models have several billions of parameters, at which point we will refer to them as *large language models* (LLMs).

Parameter-efficient Fine-tuning. When training data for a new task is available, the parameters of an LLM can be updated to minimize the error on this new task. The task itself is formatted as

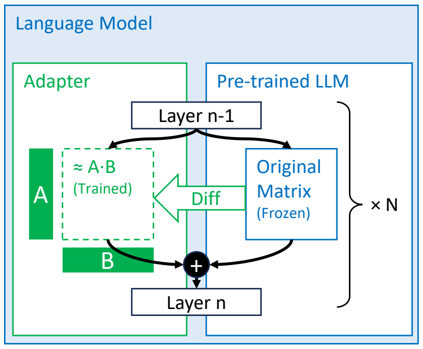


Figure 3: Low-rank adaptation (LoRA) is a parameter-efficient fine-tuning technique for LLMs that freezes the original model parameters and approximates the diff to any matrix as a product of two much smaller matrices, yielding a compact adapter that can be distributed separately and composed if needed.

sequences of tokens the model should complete. However, updating all parameters of an LLM is still prohibitively costly for us. Fortunately, we can make use of several strategies to fine-tune a model on much smaller hardware than it was initially trained on:

(1) Freeze most of the parameters and only update a subset. For example, we can freeze all parameters associated with individual tokens, retaining their pre-trained "semantics" but fine-tuning the attention mechanism. This still allows the model to learn novel patterns in the data flexibly.
(2) Represent the "diff" between original and fine-tuned parameters in (lossily) compressed form. For example, we can approximate any (full-rank) matrix as a product of two smaller (low-rank) matrices (see Figure 3). Representing the difference to each matrix in the original model as such a product of smaller matrices is the core of the LoRA (Low-rank adaptation) method [9], further reducing the number of trainable parameters during fine-tuning.
(3) Use less precision. Floating point numbers can be commonly reduced to 16, 8, or 4 bits of precision without drastically losing overall model capabilities. While we will use 16-bit floating point precision in our experiments, variants of the LoRA framework (QLoRA [2]) allow for fine-tuning 4- and 8-bit models.

A valuable aspect of the LoRA framework is that we can store and distribute the resulting "diff" as only a few megabytes of data compared to several gigabytes required to distribute the full LLM. Hence, LoRA allows us to distribute "project-specific" adapters while keeping the original model untouched and opens up the possibility to quickly swap out multiple task- or project-specific adapters during the programming workflow without having to retain separate multi-billion-parameter models on disk or in memory.

Democratization of LLMs. Despite their extremely high training costs, often including multiple terabytes of training data and several 100 000 GPU hours, many LLMs are openly available. Although the most capable models, such as OpenAI's GPT-4 [4], are only available as a service, and open-source competitors with similar performance would require expensive datacenter GPU setups, many "small LLMs" are now available. For example, *CodeLlama-7B* [16] is the smallest model of the CodeLlama architecture and fits on one high-end consumer GPU (14 GB of GPU memory in 16-bit float representation), more recent models [14] achieve comparable performance with half the memory requirements. Such models generate whole method completions of about ten lines of code within two seconds.

With the emergence of *parameter-efficient fine-tuning*, training such models for new tasks on consumer hardware has become feasible. These novel capabilities allow tool builders to incorporate LLMs locally into programming workflows without depending on an external service and associated cost and privacy concerns.

2.2 Examples in Live Programming

Live programming offers the opportunity to observe changes to a running system immediately. Since the whole program is not always running and programmers might need to explore individual aspects in isolation, introducing smaller, reproducible examples that react immediately to changes in code (and the examples themselves) carries the live programming experience over to smaller scopes such as individual methods or classes.

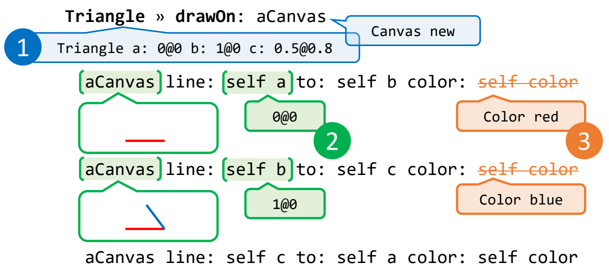
In this work, we will use the term *Example* to refer to a concrete, reproducible execution of a part of the program. In object-oriented programming, an Example for a method will consist of a concrete instance of its class with its relevant state already set and all of its arguments. According to this definition, the Example does not necessarily involve code (as in "usage example" for an API) but can be specified by a set of run-time objects and data.

Programming by Example. Examples can be used to specify the behavior of yet-unwritten code and might remain part of the programming environment to serve as future documentation. The degree of automation and immediacy ranges from manually writing code to match the example behavior, like in test-driven development (TDD), to automatically synthesizing code based on a desired exemplary outcome.

For example, *SnipPy* [5] fills in Python lines based on programmer-specified variable values or return values and keeps the example close to the code to work with both simultaneously. *Maniposynth* [7] additionally maps changes to an Example directly to changes to the underlying source code, allowing to program entirely in the example domain.

Program Comprehension by Example. When used in connection with existing code, examples allow programmers to observe the behavior of their code in concrete instances and enable exploring the consequences of a change in a concrete scenario. The more immediate feedback localized to the scenario and code of interest allows faster iterations.

An early instance of this principle is *Example-centric Programming* [3], which traces a program provided with exemplary input, illustrating the fine-grained effects of every statement on the Example. *Exemplars* [1] are constructs in the Newspeak programming system that provide concrete instances for classes and arguments



```
Triangle » drawOn: aCanvas
    aCanvas line: self a to: self b color: ~~self color~~
    aCanvas line: self b to: self c color: ~~self color~~
    aCanvas line: self c to: self a color: self color
```

Figure 4: Core elements of Babylonian Programming: The programmer-curated Example (1) provides an instance of the class and arguments for the method call to construct a full execution context. Probes (2) render dynamic data captured during the execution of the method – here, the content of the canvas – and are always updated immediately after code or Example changes. Replacements (3) allow programmers to isolate the Example from an irrelevant state by skipping the execution of an expression and proceeding as if it evaluated to the specified value.

to their methods so that code is always runnable. This allows the evaluation of arbitrary expressions inside the source code using the example data. A more recent incarnation of this principle, called Example-based Live Programming (ELP), is *Babylonian Programming* [15].

Babylonian Programming. Babylonian Programming (BP) is inspired by how ancient Babylonians expressed their algorithms - in terms of concrete examples right next to the instructions [15]. A BP-enabled programming environment introduces several concepts: (1.) the Example[2], (2.) Probes as a way to observe concrete behavior, and (3.) Replacements to override expressions with user-controlled values as illustrated in Figure 4.

In BP, an Example provides concrete values to run a particular code section and, optionally, display its final result. In object-oriented environments, this includes example instances of classes (so that a realistic value of `self` can be assumed) and example arguments needed by a method call.

A Probe can be attached to any expression, showing its value under the currently active Example(s). Probes are updated immediately on each change, i.e., the affected code path is re-executed in the background. They can use rich, domain-specific visualizations, e.g., displaying the content of a drawing buffer to help users trace its evolution.

Example Mining. A way to obtain examples is to capture them from the live environment using program instrumentation [11]. The environment needs to run the code of interest with realistic data by executing test cases or recording a realistic interaction with the program. The latter is challenging, as the user needs to start and stop the recording, and captured live object graphs can be large and highly connected compared to isolated test objects. Based on the attributes required by the code of interest, such object graphs are pruned to store only a minimized example.

3 EXAMPLE GENERATION

Our setting in this Squeak/Smalltalk-based [6] prototype envisions programmers requesting an exemplary method execution of the method they currently observe in their programming environment. The system would generate the example and run that method, automatically performing the necessary setup and instantiating arguments with an LLM's help.

We consider three phases in this framework, of which we currently implemented the first two in our prototype:

Fine-tuning We fine-tune an LLM to learn our concrete version of the standard library, best practices, and project-specific APIs. This process takes place at the beginning and once the system changes significantly.

Querying Once an example is requested, the system collects context, prepares a prompt, and passes it to the LLM. It compiles and executes the completed example.

Reification The result is reified using mechanisms like Example Mining or conversion to a Babylonian Example. From now on, the example is maintained through ELP tooling, such as Babylonian-Programming-enabled editors, and can be version-controlled.

An overview of the entire system is depicted in Figure 5.

3.1 Fine-tuning

Code-generating LLMs have a moderate understanding of Smalltalk despite mainly being trained on more common languages. However, while generating syntactically valid code, they tend to mix up different Smalltalk distributions and versions of the standard library and do not know the program for which they should generate examples. By fine-tuning, the LLM can reinforce or learn the following novel aspects:

(1) Distribution-specific best practices and standard library usage
(2) Task-specific additional syntax that we can use to inject and generate additional information beyond pure Smalltalk
(3) Project-specific coding styles and abstractions that the LLM can re-use to generate realistic examples within the program
(4) Existing examples from unit tests and code comments that the LLM can re-use and re-combine to form new examples

Challenges in Causal Language Modeling. LLMs can be fine-tuned efficiently by presenting all the source code simultaneously. E.g., given the sequence ABC with tokens A, B, and C, the completions $A \longrightarrow B$, $AB \longrightarrow C$ and $ABC \longrightarrow \langle EOS \rangle$ can be predicted and the errors back-propagated in parallel as they share most computation.

First, an essential technical constraint with the above training procedure is the limited size of the context window, which requires us to separate the training data into individual blocks[3]. For example, a class definition could end up in a different block than some of its methods, so the model does not learn that the method uses variables declared in the class. Also, existing examples (e.g., unit tests) should

[2]We capitalize the term to refer to the live Examples in Babylonian Programming rather than the generic term

[3]Our blocks turned out to be even smaller (256 tokens) than the maximum context size of the LLM (4096) before running out of GPU memory.

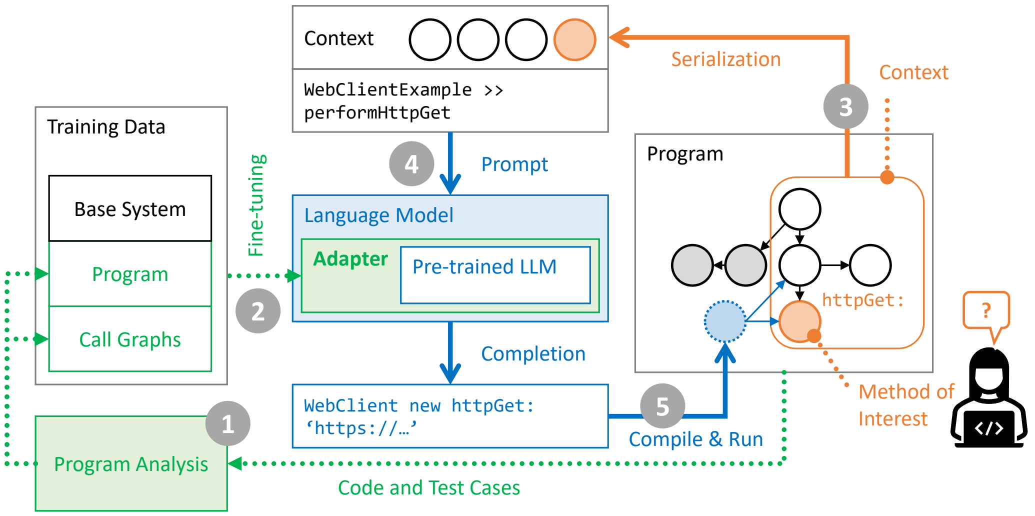


Figure 5: Interactive example generation assisted by an LLM: (1.) The base system and project are analyzed, creating training data. (2.) A pre-trained LLM is fine-tuned with that data, generating a low-rank adapter. (3.) When users request an example, the immediate dependencies of the method of interest are gathered using backward program slicing and serialized. (4.) A prompt that asks the LLM to complete an example is generated. (5.) The LLM output is compiled and run, allowing the user to inspect the dynamic behavior via debugger or ELP tools. In our prototype, the fine-tuning loop (1. and 2.) takes about an hour and runs once in preparation and then infrequently, while the interactive loop (3. – 5.) takes a few seconds.

ideally be in the same block as their exemplified method (e.g., the method under test).

Second, the left-to-right completion requires careful consideration of the order in which we present training data. For example, if we prefer completing methods that use local variables from the class definition, the methods should always follow the class definition in training data. If we prefer to generate examples for a method, a test case (an example) should follow the method under test.

Third, when trained with classes with multiple methods, the model will output a potentially unbounded number of methods and never stop by itself. Since Smalltalk has no notation to delimit classes and methods, we introduce one by re-using the LLM's existing knowledge about XML by wrapping class definitions in `<class>` ... `</class>` and methods in `<method>` ... `</method>`. Despite the unnatural mixing of XML with Smalltalk, the LLM reliably responds in this format after a few hundred examples.

Generating training data. To address the challenges of having relevant training data in proximity and the right order, we extract classes and methods from the Squeak/Smalltalk image and apply the following processing steps:

(1) Exclude methods spanning multiple blocks.
(2) Insert redundant class definitions. Smalltalk has no separate "class syntax" but creates classes calling a `subclass` method on a superclass and passing the class name, instance variables, and class variables. After every ten methods, we insert this specific call to "remind" the LLM of how the class looks.
(3) Pair callees with callers so that the LLM learns to generate code that calls previously defined methods.

We filter out any empty method, outliers with many arguments, variable and class references, deep nesting, and large test cases to improve data quality. This cleaning step leaves us with about 80% of the original methods in the Squeak/Smalltalk image as training data.

Technical implementation. We implemented the data collection in Squeak/Smalltalk. To obtain call graphs, we temporarily instrumented all methods in a package using context-oriented programming [8] to collect call graph edges, then all tests of that package are executed. Cross-package dependencies are not recorded this way, but relevant relationships within each package are captured with reasonable run-time overhead and stability compared to a full system instrumentation. We exclude kernel packages from tracing since instrumenting such low-level facilities can have unintended side effects. This code is still part of the training data but in alphabetical rather than call-graph order.

Our library exports all class and method definitions and known call graphs. We performed the data preprocessing and fine-tuning in Python. We used *CodeLlama-7B* [16] as the pre-trained model and LoRA (with rank 16) for efficient fine-tuning. We implemented the fine-tuning using PyTorch and the `transformers` and `peft` libraries and trained for two epochs on $\approx 60\,000$ methods.

Fine-tuning took less than an hour on a high-end consumer PC[4], making it feasible to re-run should the underlying system change significantly.

[4]Tested on an AMD Ryzen 7 7800X3D CPU, 64GB DDR5 memory, and NVidia RTX 4090 GPU with 24GB video memory

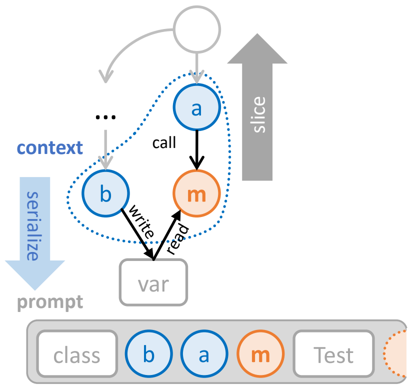


Figure 6: Prompt generation: The direct dependencies (a: call-graph dependency, b: data dependency) in a backward program slice deliver context information about preconditions to call the method of interest (m). The prompt serializes the class definition of m's class, these contextual methods, and m itself, along with a stub for a test case supposed to run m. If no such information is available, class and method definitions will still be included.

3.2 LLM Querying

The basic principle is to ask the LLM to generate a test case for the method of interest, relying on fine-tuning to incorporate knowledge about the construction of the method's receiver (`self`) and arguments. We are not interested in the assertion but in setting up an example instance of the class and calling the method, which we emphasize in the method comment. A minimal prompt can be seen in Listing 1.

```
<class>
Superclass subclass: ExemplifiedClass
"...Class definition..."
</class>

<method>
ExemplifiedClass >> exemplifiedMethod
"...Method definition..."
</method>

<class>
TestCase subclass: ExemplifiedClassTest
"...Class definition..."
</class>

<method>
ExemplifiedClassTest >> testExemplifiedMethod
"Set up an example instance of 'ExemplifiedClass'
and call method 'exemplifiedMethod'"
        [Complete from here]
```

Listing 1: Structure of a generated prompt. Class and method definitions are taken from the Smalltalk image, while the test class is generated to match the class and method names. Completion will generate a short unit test.

The fine-tuned background knowledge, however, can become outdated and is subject to hallucinations (e.g., an argument named `request` might be filled with the result of the call `Request new`, although the class is named `WebRequest`). Fortunately, LLMs tend to imitate and repeat patterns occurring in their prompt. This characteristic allows us to reduce hallucinations by incorporating context that demonstrates how the method is called in practice, including the necessary set-up code.

For methods with associated call graph data (constructed during fine-tuning), we select actual callers and methods that set variables required by the method of interest. Outside call graph coverage, e.g., for novel methods, we heuristically identify callers as methods in the same class referencing the method's name. The context selection and serialization procedures are illustrated in Figure 6.

In the Smalltalk image, the LLM response is rewritten to represent a test case with a specific category for generated tests (thus not polluting the existing test suite) and a specific superclass where assertion methods are overridden to prevent failures. The compiled test case is run. At this point, we can integrate future tooling: An active example miner can record the live objects instantiated by the generated test and attach them as Babylonian Examples to the method.

3.3 Usage Example

The preliminary prototype that implements this functionality in the Squeak/Smalltalk browser is illustrated in Figure 7. For now, a simple button triggers the above generation cycle. In this example, the method implementing an HTTP GET request is used as method of interest, and a breakpoint (`self halt`) has been inserted to summon a debugger that can inspect the LLM-generated state. The code generated by the LLM is shown in Listing 2. Note that the `WebClient` implementation only has integration tests that test against an isolated mock server. A test case similar to the one generated is not included in the training data but might have been inferred from the method comment. The fact that the LLM uses a block to handle the request object could be inferred from the context, which included a caller to that method.

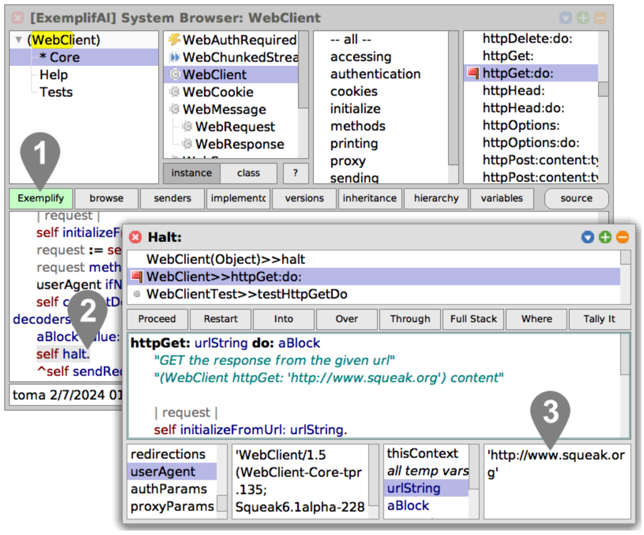
```
WebClientTest >> testHttpGetDo
"Set up an example instance of 'WebClient' and
call method 'httpGet:do:'."

| client |
client := WebClient new.
client httpGet: 'http://www.squeak.org'
        do: [:request |
                self assert: request method = 'GET'].
```

Listing 2: Test example generated by our fine-tuned CodeLlama-7B

4 DISCUSSION AND OUTLOOK

This prototype demonstrates that local LLM fine-tuning and example generation have become feasible on consumer hardware, even for less prevalent environments like Squeak/Smalltalk. It also shows how many moving parts and design decisions must be made to integrate generative AI into a programming workflow.



```
WebClient(Object)>>halt
WebClient>>httpGet:do:
WebClientTest>>testHttpGetDo
```

```
httpGet: urlString do: aBlock
    "GET the response from the given url"
    "(WebClient httpGet: 'http://www.squeak.org') content"

    | request |
    self initializeFromUrl: urlString.
```

```
redirections     'WebClient/1.5             thisContext       'http://www.squeak.or
userAgent        (WebClient-Core-tpr        all temp vars     g'
authParams       .135;                      urlString
proxyParams      Squeak6.1alpha-228         aBlock
```

Figure 7: Screenshot of our initial prototype. The Smalltalk browser has selected the `httpGet:do:` method of the `WebClient` class. (1) A novel "Exemplify" button initiates example generation. The button color indicates whether an example has previously been generated successfully and can be re-run (green) or would be generated first (red). (2) We inserted a breakpoint into the method. (3) Clicking the "Exemplify" button triggers the breakpoint and summons a debugger. We see that the LLM has chosen to call the method with the example URL that often appears in code comments. The call stack shows the generated test case.

4.1 Limitations

Our current approach to using a small, fine-tuned open-source LLM brings several yet unsolved challenges and limitations. The quality of the output hinges on many factors. We noticed that any of the following decisions can have a significant impact on the capabilities of the LLM-based tool:

- Selection, formatting, ordering, and augmentation of training data
- Hyperparameters, such as learning rate, size of the low-rank matrices, and number of epochs
- The selected method to generate code from the LLM’s probability distribution (e.g., greedy, sampling, or beam search).
- The method to determine when to stop generation
- Special syntax used for fine-tuning and prompting
- Selection of relevant context
- Formulation and generation of prompts

Since the system as a whole is responsible for the programmers’ experience, optimizing these aspects in isolation is hard.

For example, a more "logical" ordering of training data might result in a better-trained LLM as judged by loss metrics. However, the model might generalize poorly from the provided context when used in practice. On the other hand, additional syntax like the introduced XML tags makes it easier to stop generation and helps the model learn the output format we expect, but will inevitably increase loss during training since XML is rarely combined with Smalltalk code. A significant challenge that will remain is the selection of proper context to generate meaningful examples. We found through experimentation that the neighboring nodes in a program slice are helpful as they show how a method can be called. The model can generate working examples without this data, although with a higher chance of misinterpreting argument names and hallucinating "shortcuts" to fill them.

Our current training data augmentation relies on test coverage to simplify program analysis and slicing. However, when the code base is sufficiently covered, methods like Example Mining are less error-prone and preferable over generative AI. The general principle, however, will likely work with static analysis in the absence of tests.

4.2 Evaluation

An essential next step to address the abovementioned challenges is creating an evaluation framework for LLMs in live programming environments to allow systematic experimentation with each variability source.

We plan to obtain evaluation data from the broader Smalltalk ecosystem and categorize methods by the complexity of the required example (primitive data, nested/structured data, instantiated objects, complex object graphs, and callbacks) to gradually test and extend our framework’s capabilities. While speed and correctness can be measured by executing generated examples on the benchmark dataset, utility and explanatory power are much harder to operationalize.

We first plan to contrast generated examples with examples mined from actual usage and unit tests, quantifying how closely the generated object graphs resemble recorded ones. A second step would employ a more capable LLM (e.g., GPT-4 or a smaller LLM specifically fine-tuned to judging examples) to evaluate the generated examples. This evaluation method is known as *LLM-as-a-judge* [19]. Human subjects should evaluate a subset of examples in a controlled study.

Apart from using GPT-4 to pilot and scale up a "user" study, we will compare our outputs to those generated by GPT-3.5 and GPT-4 - based on initial experiments (see Figure 8), we expect more comprehensive and illustrative examples with the trade-off of encountering more hallucinations due to mixing up Smalltalk distributions and versions.

4.3 Integration and Interaction

Multiple Examples. Our prototype currently generates a single example. The next step will be the generation of multiple examples in parallel. The overhead is negligible: 16 parallel generations only take about 50% longer than a single generation due to how GPUs reuse cached LLM parameters. Programmers can subsequently select among several answers, reducing the risk of the single answer being faulty or unhelpful.

Moreover, we can generate based on different prompts in parallel, e.g., including different contexts in each, to obtain a variety of examples and increase the chance that one is very useful. How to select the most useful one for the user automatically remains an open question.

System: You generate usage examples that help understand source code given in Squeak/Smalltalk. The user provides the source code to be illustrated. You answer with executable source code in the Smalltalk language that calls the method provided by the user with a realistic example.

User: (Provides the `WebClient >> httpGet:do:` method)

Assistant: Here is an example code demonstrating how you would use the `` `httpGet:do:` `` method:

~~~
```Smalltalk
| webClient |
webClient := WebClient new.
webClient httpGet:
  'https://jsonplaceholder.typicode.com/posts/1'
  do: [:request |
    Transcript show: request asString; cr]
```
~~~

Figure 8: Using GPT-4's chat API for example generation (line breaks inserted manually to fit the paper).

Babylonian Programming. The framework is currently independent of any implementation of Example-based Live Programming. However, we aim to integrate LLM-based example generation into Babylonian Programming to extend the scope beyond manually curated Examples.

Additionally, we can use the LLM to interactively refine existing Examples or use them as context to improve example generation for new code passages. We will further explore ways to use programmer-placed BP elements (such as Probes) to inform the LLM about important context or places to be reached by the generated example. The execution of BP Examples yields valuable tracing data for further refining our context selection.

Automated Re-Fine-Tuning. A fine-tuned LLM will eventually become outdated as the software evolves. However, new code becomes available when programmers work with the software, and live objects and user interaction can be recorded and used for training.

Our prototype was limited to a cold start by only relying on source code and automated dynamic analysis, but manually executing the program (e.g., small experiments done in a REPL/workspace, objects encountered during debugging, etc.) and the users' behavior (co-changed methods or navigation between code locations) provide valuable contextual data and thus opportunities for improved fine-tuning and prompting.

External Data and Documentation. Examples can originate from outside the programming environment. Mailing lists, data on collaborative development platforms (e.g., GitHub issues, pull requests, and discussions), documentation, and requirements are possible sources of examples. They can be provided as additional context to the LLM to make them executable. The *TestPilot* system [17], for example, uses usage examples from documentation to generate tests.

Retrieving the most relevant data from a corpus of such documents is an information retrieval problem, and generating from such a context is known as *retrieval-augmented generation* (RAG).

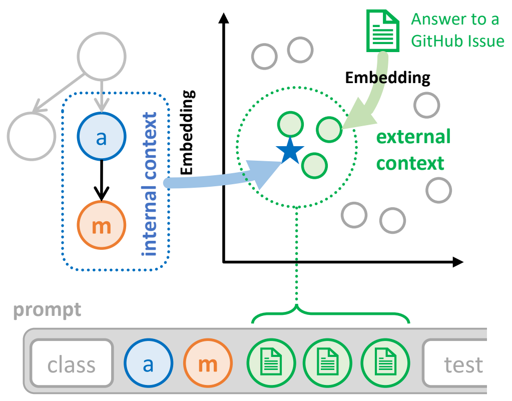


Figure 9: Using retrieval-augmented generation (RAG) to pull examples from external data sources. The relevance of each item is scored by its proximity to the original context (here, the method of interest `m` and its caller `a`) in the embedding space. The most relevant items are included in the LLM prompt.

Current implementations use embedding functions to establish semantic similarity. They project both the initial prompt and all available information items into the same high-dimensional vector space (the latter being done once ahead of time), and then the nearest items are selected as context (see Figure 9 for an augmented example based on Figure 6). The embedding function can be fine-tuned to the domain or task context if positive and negative samples (e.g., a prompt, a related item, and an unrelated item) are available.

While RAG can be a powerful addition to our framework, more design decisions have to be made, such as the choice of embedding function and the granularity of external items (should we embed and retrieve whole GitHub issues or split them into smaller chunks?). Also, the same problems that influence the quality of the prompt (e.g., selecting project-internal context) influence what we can retrieve externally.

Considerations when Using External Data in LLMs. We must consider various ethical and legal implications when incorporating open-source community content whose creators rarely benefit from or know of their involuntary contribution to LLM training data. These could involve ensuring proper attribution, obtaining consent, respecting opt-outs, complying with possibly mixed licenses, and contributing back to the community.

Although the generated code is mainly invisible to the user of our system, LLMs still have a chance to reproduce content close to some original training data, thereby subjecting its users to plagiarism- and copyright-related uncertainty.

CONCLUSION

The capability of large language models to generate code and recent developments in fine-tuning and minimization made them a relevant component in building tools for programming activities.

We demonstrated how we can leverage a locally hosted LLM to assist programmers in the Squeak/Smalltalk live programming environment in obtaining runnable examples for code they are attempting to understand. The proposed integration into Babylonian Programming promises further program comprehension and programming education benefits.

The surprising number of design decisions in such a system, ranging from fine-tuning data over context selection to prompt engineering to control the somewhat unpredictable behavior of pre-trained LLMs, is both a challenge for tool developers and an opportunity to build highly specialized tools for an improved programming experience. Our proposed framework serves as a starting point for a more systematic and holistic evaluation of these design decisions in the context of example generation and program comprehension.

ACKNOWLEDGMENTS

This work is supported by the HPI-MIT "Designing for Sustainability" research program[5]. We also gratefully acknowledge the financial support of HPI's Research School.

REFERENCES

[1] Gilad Bracha. 2021. Enhancing Liveness with Exemplars in the Newspeak IDE. https://newspeaklanguage.org/pubs/newspeak-exemplars.pdf.

[2] Tim Dettmers, Artidoro Pagnoni, Ari Holtzman, and Luke Zettlemoyer. 2023. QLoRA: Efficient Finetuning of Quantized LLMs. https://doi.org/10.48550/arXiv.2305.14314 arXiv:2305.14314 [cs]

[3] Jonathan Edwards. 2004. Example Centric Programming. *ACM SIGPLAN Notices* 39, 12 (Dec. 2004), 84–91. https://doi.org/10.1145/1052883.1052894

[4] OpenAI et al. 2023. GPT-4 Technical Report. https://doi.org/10.48550/arXiv.2303.08774 arXiv:2303.08774 [cs]

[5] Kasra Ferdowsifard, Allen Ordookhanians, Hila Peleg, Sorin Lerner, and Nadia Polikarpova. 2020. Small-Step Live Programming by Example. In *Proceedings of the 33rd Annual ACM Symposium on User Interface Software and Technology (UIST '20)*. Association for Computing Machinery, New York, NY, USA, 614–626. https://doi.org/10.1145/3379337.3415869

[6] Adele Goldberg and David Robson. 1983. *Smalltalk-80: The Language and Its Implementation*. Addison-Wesley.

[7] Brian Hempel and Ravi Chugh. 2022. Maniposynth: Bimodal Tangible Functional Programming. In *DROPS-IDN/v2/Document/10.4230/LIPIcs.ECOOP.2022.16*. Schloss-Dagstuhl - Leibniz Zentrum für Informatik. https://doi.org/10.4230/LIPIcs.ECOOP.2022.16

[8] Robert Hirschfeld, Pascal Costanza, and Oscar Nierstrasz. 2008. Context-Oriented Programming. *Journal of Object Technology, March-April 2008, ETH Zurich* 7, 3 (2008), 125–151. https://doi.org/10.5381/jot.2008.7.3.a4

[9] Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. 2021. LoRA: Low-Rank Adaptation of Large Language Models. https://doi.org/10.48550/arXiv.2106.09685 arXiv:2106.09685 [cs]

[10] Donald E. Knuth. 1972. Ancient Babylonian Algorithms. *Commun. ACM* 15, 7 (July 1972), 671–677. https://doi.org/10.1145/361454.361514

[11] Eva Krebs, Patrick Rein, and Robert Hirschfeld. 2022. Example Mining: Assisting Example Creation to Enhance Code Comprehension. In *Companion Proceedings of the 6th International Conference on the Art, Science, and Engineering of Programming (Programming '22)*. Association for Computing Machinery, New York, NY, USA, 60–66. https://doi.org/10.1145/3532512.3535226

[12] Ziyang Luo, Can Xu, Pu Zhao, Qingfeng Sun, Xiubo Geng, Wenxiang Hu, Chongyang Tao, Jing Ma, Qingwei Lin, and Daxin Jiang. 2023. WizardCoder: Empowering Code Large Language Models with Evol-Instruct. *arXiv preprint arXiv:2306.08568* (2023).

[13] Kasia Muldner, Jay Jennings, and Veronica Chiarelli. 2022. A Review of Worked Examples in Programming Activities. *ACM Transactions on Computing Education* 23, 1 (Dec. 2022), 13:1–13:35. https://doi.org/10.1145/3560266

[14] Nikhil Pinnaparaju, Reshinth Adithyan, Duy Phung, Jonathan Tow, James Baicoianu, and Nathan Cooper. 2024. Stable Code 3B. https://huggingface.co/stabilityai/stable-code-3b

[15] David Rauch, Patrick Rein, Stefan Ramson, Jens Lincke, and Robert Hirschfeld. 2019. Babylonian-style Programming - Design and Implementation of an Integration of Live Examples Into General-purpose Source Code. *Art Sci. Eng. Program.* 3, 3 (2019), 9. https://doi.org/10.22152/programming-journal.org/2019/3/9

[16] Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, Jade Copet, Faisal Azhar, Hugo Touvron, Louis Martin, Nicolas Usunier, Thomas Scialom, and Gabriel Synnaeve. 2024. Code Llama: Open Foundation Models for Code. https://doi.org/10.48550/arXiv.2308.12950 arXiv:2308.12950 [cs]

[17] Max Schäfer, Sarah Nadi, Aryaz Eghbali, and Frank Tip. 2024. An Empirical Evaluation of Using Large Language Models for Automated Unit Test Generation. *IEEE Transactions on Software Engineering* 50, 1 (Jan. 2024), 85–105. https://doi.org/10.1109/TSE.2023.3334955

[18] Christoph Thiede, Marcel Taeumel, and Robert Hirschfeld. 2023. Time-Awareness in Object Exploration Tools: Toward In Situ Omniscient Debugging. In *Proceedings of the 2023 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward! 2023)*. Association for Computing Machinery, New York, NY, USA, 89–102. https://doi.org/10.1145/3622758.3622892

[19] Lianghui Zhu, Xinggang Wang, and Xinlong Wang. 2023. JudgeLM: Fine-tuned Large Language Models Are Scalable Judges. https://doi.org/10.48550/arXiv.2310.17631 arXiv:2310.17631 [cs]

Received 2024-02-08; accepted 2024-02-26

[5]https://hpi.de/en/research/cooperations-partners/research-program-designing-for-sustainability.html