Supporting Web-based API Searches in the IDE Using Signatures

Nick C. Bradley
ncbrad@cs.ubc.ca
The University of British Columbia
Canada

Thomas Fritz
fritz@ifi.uzh.ch
University of Zurich
Switzerland

Reid Holmes
rtholmes@cs.ubc.ca
The University of British Columbia
Canada

ABSTRACT

Developers frequently use the web to locate API examples that help them solve their programming tasks. While sites like Stack Overflow (SO) contain API examples embedded within their textual descriptions, developers cannot access this API knowledge directly. Instead they need to search for and browse results to select relevant SO posts and then read through individual posts to figure out which answers contain information about the APIs that are relevant to their task. This paper introduces an approach, called Scout, that automatically analyzes search results to extract API signature information. These signatures are used to group and rank examples and allow for a unique API-based presentation that reduces the amount of information the developer needs to consider when looking for API information on the web. This succinct representation enables Scout to be integrated fully within an IDE panel so that developers can search and view API examples without losing context on their development task. Scout also uses this integration to automatically augment queries with contextual information that tailors the developer's queries, and ranks the results according to the developer's needs. In an experiment with 40 developers, we found that Scout reduces the number of queries developers need to perform by 19% and allows them to solve almost half their tasks directly from the API-based representation, reducing the number of complete SO posts viewed by approximately 64%.

CCS CONCEPTS

• **Software and its engineering** → **Software maintenance tools**; *Integrated and visual development environments*; • **Human-centered computing**;

KEYWORDS

API signatures, code search, controlled experiment

ACM Reference Format:
Nick C. Bradley, Thomas Fritz, and Reid Holmes. 2024. Supporting Web-based API Searches in the IDE Using Signatures. In *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE '24), April 14–20, 2024, Lisbon, Portugal.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3597503.3639089

ICSE '24, April 14–20, 2024, Lisbon, Portugal

ACM ISBN 979-8-4007-0217-4/24/04…$15.00
https://doi.org/10.1145/3597503.3639089

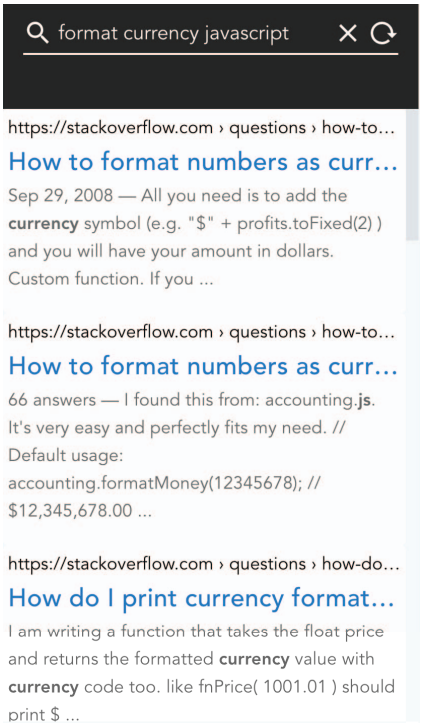
1 INTRODUCTION

Developers rely on the specialized functionality provided by application programming interfaces (APIs) to help them complete their development tasks [44]. Before they can use an API, developers need to discover which APIs might be appropriate considering the code they are working on and how the API is meant to be used [1]. One common way developers look for potential APIs is through search engines like Google [16, 45]. Unfortunately, this introduces workflow friction as developers have to frequently switch between their IDE and browser [6, 32], copy code terms into their searches [21, 46], and manage a large number of browser tabs as they assess different solutions [9, 25]. Even when a web page, such as a Stack Overflow (SO) post, contains the required API information, developers have to read through the content to identify which APIs are present and how they might fit into their code [17]. These extra steps affect developers' focus and increase their cognitive load [12].

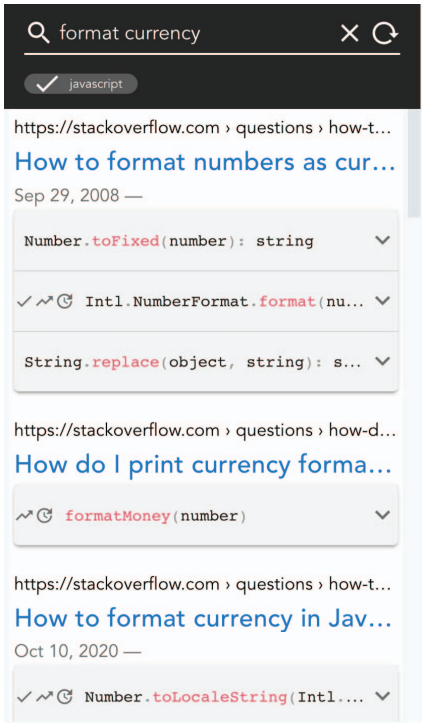
To address some of these challenges, researchers have investigated approaches to help developers find information relevant to their development tasks. Early web-based code search approaches, such as Mica [42] and Assieme [14], augmented search results with links to code elements contained within the search result pages to help developers refine the results. With the growth of SO as an essential source of development knowledge, approaches have focused on summarizing SO posts to answer a developer's query. For example, AnswerBot uses natural language processing to combine the textual information from multiple posts to create diverse summaries without focusing specifically on providing API information [46]. Biker provides summary documentation for specific APIs relevant to a developer's search query but requires a custom search engine indexed on SO data dumps and the official Java documentation [17]. More recent tools based on large language models (LLMs), such as ChatGPT and Copilot, offer alternative ways for developers to obtain solutions to their tasks. However, the solutions produced by these approaches can be incorrect [30], verbose [18], and often require repeated refinement of the queries [28], making it difficult for developers to understand and compare solutions.

In this paper, we introduce an approach, called Scout, which succinctly represents the results of a search query using API signatures to make it easier for developers to identify and compare information relevant to their API search tasks (Figure 1b). Specifically, Scout adapts the presentation of SO Google search result pages so developers can more directly compare alternative solutions in the limited space of the IDE by hiding non-essential and duplicate information.

Scout extracts these *signatures*, consisting of a method name, class type, parameter type(s), and return type, from the SO posts in the developer's search results. These signatures are presented as focus points, giving developers an overview of potential solutions



(a) Standard view with Google's text preview.



(b) Scout with API signatures.

(c) Expanded API call signature in Scout with class type (`Intl.NumberFormat`), function name (`format`), parameter types (`number`) and return type (`string`). Icons from left to right indicate whether the signature came from the *accepted*, *top-voted*, or *most recent* answer. The currently selected USAGE tab shows just the function call and the parent variable (`money`) while the CODE tab shows the full code example. Clicking the icon on the right opens the full answer inside Scout.

Figure 1: Search results interface.

without having to examine each post individually. This overview encourages top-down information gathering where developers first identify the most appropriate signatures before examining detailed code examples and descriptions from the source SO posts.

Scout is integrated into the IDE enabling developers to assess their search results directly from their code without losing focus on their current task. Using this integration, Scout also automatically derives terms, such as variable types and library names, from the code the developer is working on as context to improve the API signature recommendations. When making a search, Scout suggests some of these terms to help scope the developer's query. Once the developer has performed their search, Scout extracts the embedded API call signatures from each SO post in the search results. Scout presents the top-three signatures under the title of each post after ranking the signatures by their frequency of occurrence and the number of types they share with the developer's context.

To evaluate Scout, we conducted a controlled experiment with 40 developers. In the experiment, we asked participants to complete six coding tasks, randomly assigning them to either our experimental treatment, which presented API call signatures, or a baseline treatment which presented regular Google search results within the IDE. For the study, we implemented Scout as a VSCode IDE extension that ran directly in participants' browsers, recording their interaction and feedback as they completed the study tasks.

Overall, Scout made the most relevant information more directly accessible, resulting in participants completing almost half of the tasks directly using Scout's signature presentation and significantly reducing the number of posts opened by participants by 64%. Further, participants valued Scout's API-centered presentation and Scout's automatically inferred context terms helped to significantly reduce the number of searches that developers performed.

This paper makes two primary contributions:

(1) A novel approach and prototype tool supporting developers' API queries by (a) presenting the search results centered around API signatures to direct further investigation, and (b) using type information to rank the fit of the API signatures with the developer's code. Unlike prior approaches, Scout works with live search results rather than building a custom search engine indexing offline data dumps.
(2) A dataset and empirical findings from a controlled experiment with 40 developers demonstrating that Scout supports developers in finding the information they need to complete API tasks more effectively than a traditional web search.

2 APPROACH

Up to 50% of developers' searches are to locate and obtain API information relevant to their current task [1, 14, 16, 21, 39, 45]. For these searches, developers usually switch from their IDE to a web browser, create a query, parse through the query results, go through the result pages to locate relevant information, and, once located, go back and forth between their code and the web browser to understand if and how the information fits [6, 31]. These actions are extraneous to the task and detrimental to developers' focus and cognitive load [12].

Our objective is to direct developers to the API information most relevant to their tasks. Concretely, we aim to focus a developer's search and navigation towards task-relevant information, reduce the amount of content developers need to investigate, tailor the remaining information to the developer's working context, and present the results within the IDE to minimize cross-application switches.

We chose a focus+context design which enables developers to first identify interesting APIs and then obtain additional contextual information to help the developer integrate the API into their code [26]. We use API call signatures as focus points within the results of a search (Figure 1b) since call signatures are both compact and provide enough information for developers to understand how they work [24]. Developers can expand the API signature to a minimal code example to obtain additional usage context (Figure 1c).

The focus+context design was the result of an iterative design process informed by our prior observations of developers searching Google during live-streamed development sessions [6]. Our initial design used text snippets, extracted from SO answers referencing elements in the developer's source code, to explain the API. However, we found that prose made it difficult for developers to identify how to use the API (e.g., input and return types) and often required considerable space in the IDE due to the low information density of written text. In a subsequent design, we provided minimal code examples, displaying only the lines of code referencing the API related to the developer's source code, to make API usage more apparent. However, even with this design, it was still difficult for developers to identify and compare APIs as the API calls were obfuscated by the other code in the example. Nonetheless, the design provided valuable context when developers wanted to use the API, so we included it in our prototype as the default view when the signature is expanded.

Our final Scout prototype is implemented as VSCode extension supporting the JavaScript programming language and focuses on retrieving API-related information from Stack Overflow as it is one of the most commonly used Q&A websites for developers [45]. Figure 2 provides an outline of the steps Scout performs when answering a developer's search query.

2.1 Contextualizing Developer's Searches

Scout uses context terms extracted from the code the developer is working on in the IDE's editor to (i) automatically augment the developer's search queries and (ii) rank the search results. Whenever a developer switches to Scout, the context terms are extracted from the Abstract Syntax Tree (AST) of the developers' active code file. Scout identifies the function the developer is working on (the *active function*) by using the cursor location in the code file. Scout retrieves the function's *class type*, *parameter types*, *return types*, and the types of any in-scope variables, along with the names of external libraries and any function calls made within the active function whose call chain originates in an external library. Scout retrieves the programming language from the file extension.

Scout automatically suggests the programming language, external library names, and library call context terms whenever a developer performs a query (Figure 2a). By default, these terms are added to the developer's query to tailor the search to the developer's code

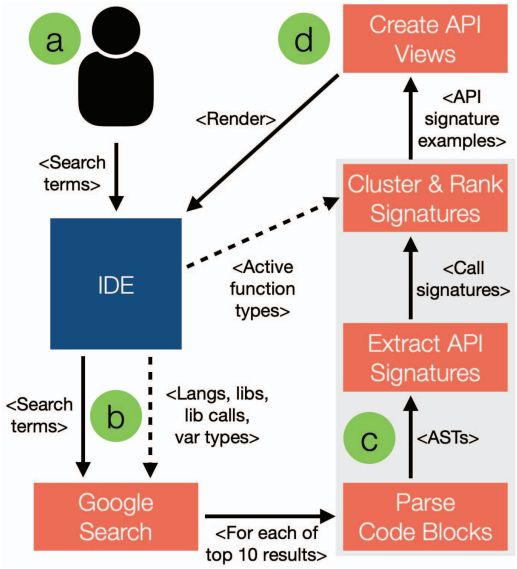


Figure 2: Scout's search process.

context. Adding these context terms to a developer's search query is motivated by prior observations that developers typically include terms such as the name of the programming language, frameworks, and libraries (e.g., HTML/JQuery) when searching the web [16]. The developer's search is ultimately performed using Google[1] with the `site:stackoverflow.com` domain filter (Figure 2b).

When recommending API signatures identified within the search results (Section 2.2), Scout uses the type context (class, parameter/variable, and return types) to rank the signatures based on how easily they could be implemented in the scope of the active function.

2.2 Identifying Relevant Signatures

To generate the API signatures from SO posts, Scout uses a three-step process (Figure 2c). First, the code blocks within each answer of a SO post are merged into a single virtual file which is then parsed into an AST using `ts-morph`,[2] a wrapper for the TypeScript compiler. When parsing, Scout assumes that the programming language in the code blocks is the same as that of the source code the developer is working on in the IDE; answers for which parsing fails are ignored. Second, Scout identifies the top-level call expressions within the parsed ASTs, excluding commonly used global calls, which, for JavaScript, include `console.*`, `alert`, and `require`. Finally, Scout uses a best-effort approach to resolve the class, parameter, and return types for the signatures based on the calls and surrounding example code in the SO answer. Specifically, Scout infers the type from literal values used in the example. In cases where answers are incomplete (e.g., an undefined variable passed as a parameter), Scout extracts a partial signature omitting the types that could not be determined (for parameter types, we substitute `unknown` to maintain parameter positions). For calls that

[1]We use SerpAPI (https://serpapi.com) to avoid manually scraping search results.
[2]https://ts-morph.com/

are part of the language, Scout further abstracts any literal types using the compiler's built-in type definitions.

Scout includes the top three API call signatures generated for a SO post, based on the signature's frequency within the post and by the number of types that overlap with the function the developer is actively working on (see section 2.1). Types overlap when the signature's class type matches an imported type or any types in the scope of the developer's active function (e.g., variable or parameter), the signature's parameter types match any types in the scope of the active function, or the signature's return type matches the return type of the active function.

2.3 Integrating Results Within the IDE

Scout renders a succinct summary of a developer's search results (Figure 2d), providing the title of the post (textual description) and the top three API call signatures and code examples essential for understanding and using the APIs [29, 47]. In contrast to Google where the information provided by the summary varies (Figure 1a), Scout's signature summary consistently provides the required information (Figure 1b). Additionally, Scout prefixes the signatures with up to three icons (check mark, arrow, and clock) indicating the quality of the SO answers (accepted, top-voted, and most-recent, respectively) the signature comes from. The compact nature of Scout's signature representation, combined with the textual description, make the representation ideal for summarizing search results within the limited space in the IDE.

To help developers better assess the suitability of an API call and integrate it into their code, Scout also provides a minimal usage example when the signature is expanded (Figure 1c). Scout extracts usage examples from the highest voted answers that contain the signature. For the minimal usage example, Scout includes only the function call, source object, parameter values, and the return value of the call. Scout also provides the corresponding full code example in the CODE tab (Figure 1c) from the answer of the post, in case developers need additional information to understand the API call and its context. Finally, developers can open the full answer on the SO post within Scout by clicking on the signature.

Our approach aims to reduce the effort required for locating and integrating relevant API information into the code. Specifically, to avoid the back-and-forth navigation between the IDE and pages in a web browser, Scout presents the search results next to the developer's active code editor and focuses on the most relevant information to accommodate the limited space within the IDE. The proximity of the search results to the active code file, and the focus on the relevant parts of the search results within Scout, are meant to make it easier for developers to understand if an API call is suited for their coding task while allowing the API call to be used directly.

3 METHODOLOGY

We conducted a controlled experiment with 40 participants comparing our signature-based presentation with standard Google search results to evaluate whether Scout effectively distills the essential information developers require to identify API solutions.

3.1 Experimental Design

The controlled experiment followed a within-participant design in which participants completed tasks with two treatments: a control treatment that presented results directly from Google search and our experimental interactive API treatment. Participants were randomly assigned to one of four counterbalanced experimental blocks, as shown in Figure 3. The experiment was conducted in three phases across the six independent tasks described in Table 1.

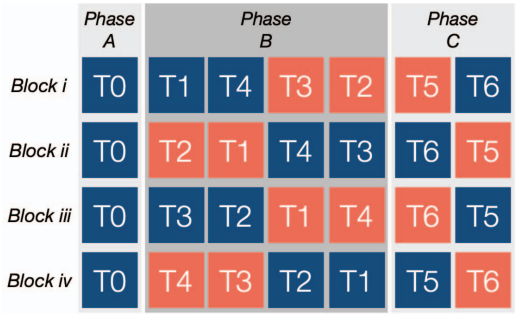


Figure 3: Randomized blocking for the tasks T0–T6. Dark tasks: Scout treatment; light tasks: control treatment.

Phase A: Training. At the outset of the experiment, we introduced participants to Scout through a guided walkthrough where they completed a mandatory tutorial task (T0).

Phase B: Controlled queries. In this phase, we asked participants to solve four independent tasks (T1–T4), two with the control (Google search) and two with the experimental treatment (Scout). While the task and treatment orders were randomized, treatment blocks were always consecutive to reduce disorientation caused by changing the interface too often. In this phase, we suggested an initial query for each task so that participants could focus on the presentation of the results without the difficulty of choosing effective query terms [19, 35, 36]. We constructed the initial queries based on the most common query format used by developers, consisting of the language, a verb, and compliment terms, and verified that the results contained a solution for the task [16]. The initial query also served as a control ensuring that participants were given the same initial set of search results under both treatments. However, participants were free to make additional searches if they desired.

Phase C: User queries. The final two tasks (T5–T6) simulated scenarios where participants needed to formulate and refine their search queries, as they would when completing their own development tasks. As with Phase B, participants were able to revise and search as many times as they thought necessary to find information that could help them complete their task. The initial treatment in Phase C was aligned with Phase B to reduce treatment disorientation.

After each task in Phase B and C, participants were given a short two-question survey to reflect on the task and to take a break before continuing. While each task could be solved in multiple ways, the simplest solution for each involved API calls. Participants completed the tasks by implementing the task requirements in individual methods following a test-driven approach using the provided unit tests to check their solutions. Participants had to pass all unit tests in the time allotted for each task (Table 1) before

they could progress. Participants were notified when one minute remained; if one or more tests were still failing once the time had elapsed, participants were prompted to immediately continue to the next task.[3] The one exception was the tutorial task T0, for which there was no time limit and participants had to pass all tests.

Table 1: Task descriptions. Suggested keywords include both context terms (in bold) and search terms (T0–T4).

Task	Time	Description/*Suggested Keywords*
T0		Sum numeric property in object array. ***javascript*** *sum object property array*
T1	6 min	Clone array of objects. ***javascript*** *clone array*
T2	6 min	Sort array of objects by property. ***javascript*** *sort array of objects descending*
T3	6 min	Find object in array with a specific value. ***javascript*** *find item in array by value*
T4	6 min	Localize currency display format. ***javascript*** *format currency*
T5	7 min	Compute a start date given end date + interval. ***javascript moment***
T6	7 min	Create endpoint to serve a PDF report. ***javascript express path***

The study was designed to take between 60 and 90 minutes to complete, and to be executed entirely within the participant's browser. Participants were able to begin the study at any time but had to complete it within 180 minutes. To begin the study, participants followed an anonymous link to our online survey which provided details about the study including the number of tasks, and the overall time and JavaScript experience required. If the participant consented, they were randomly assigned to one of the four treatment blocks for the remainder of the study. We automatically provisioned a GitHub repository, started an online VSCode IDE (Codespace) instance, and provided participants with instructions to access this IDE in their browser.

The web-based IDE for the study is shown in Figure 4. The main Scout view is shown in Figure 4a; participants could enter query terms (Figure 4i), view or disable automatically-provided context terms (Figure 4ii), and view the API call signatures (Figure 4iii) adjacent to their code (Figure 4b). Instructions for each task were shown in a dedicated pane adjacent to the source code (Figure 4c). Participants could click the `Done` button once the tests passed (or the timer expired). Clicking `Done` replaced the instructions with the task feedback survey and a link to the next task. The time remaining was shown in the status bar at the bottom of the window (Figure 4d). Once participants completed the final task they were directed to an exit survey about their overall experience and demographics.

3.2 Participants

We advertised the study on a variety of online platforms including Twitter, Mechanical Turk,[4] and Prolific,[5] in addition to directly contacting potential participants through personal contacts. Workers had to demonstrate their development knowledge by successfully completing a screening questionnaire to be eligible to participate. Figure 5 outlines our recruitment process for the study.

On Prolific, 455 workers had profiles listing appropriate development experience [38], and passed an established pre-screener which involved answering five basic development questions in a two-minute survey [10]. As we were not sure how many workers would actually complete the study, we decided to invite the top 100 qualified workers based on the number of studies they previously completed on the platform. We also invited one additional worker who requested to take part out of interest. Ultimately, we received responses to 66 of the 101 invitations sent. On Mechanical Turk, 413 workers opened our survey. Since Turk does not have a separate pre-screening and invitation process, workers had to complete the screening questions at the start of the survey before they could attempt the study. On both platforms, participants who did not complete the six study tasks (either successfully or by timing out) were considered to have withdrawn from the study.

Before conducting the full experiment, we took steps to ensure the quality of our experimental procedure. First, we piloted the study with 20 Mechanical Turk workers. We simplified the tasks so they could be completed in the allotted time, modified two survey questions to elicit more direct responses, and added checks to ensure that participants were actual developers and that they attempted all of the tasks. Second, we estimated the number of participants required to observe differences between the two treatments. We calculated that we needed a minimum of 22 participants, contributing 66 data points over the three tasks in each condition, using a t-test power analysis with a power level of 0.8, significance level of .05, and recommended Cohen's d effect size of 0.5 [11].

In total, we received 40 complete responses from 21 professional developers, 10 students, and 9 individuals who work with code in other capacities. Participants (31/40 male, 7/40 female, 2/40 transgender and nonbinary) reported an average of 5.8 ± 5.9 years of professional experience and 3.4 ± 4.3 years of JavaScript experience. The majority of participants came from Prolific (28/40) with the remaining from Mechanical Turk (7/40), Twitter (1/40), and through direct contact (4/40). Participants were paid $25.

3.3 Data Collection

The 40 participants completed 240 tasks in total. Participants were equally assigned to each of the four trial groups, resulting in 20 observations per task (T1–T6) and treatment (control and Scout). We consider all of these observations in the following analysis, including 67 cases (34 control + 33 Scout) where participants exceeded the time allotted for the task. However, we exclude 11 cases (6 control + 5 Scout) where participants did not open the provided search interface and focus on the remaining 229 observations.

[3] However, participants could dismiss the prompt and continue working.
[4] https://www.mturk.com
[5] https://www.prolific.co

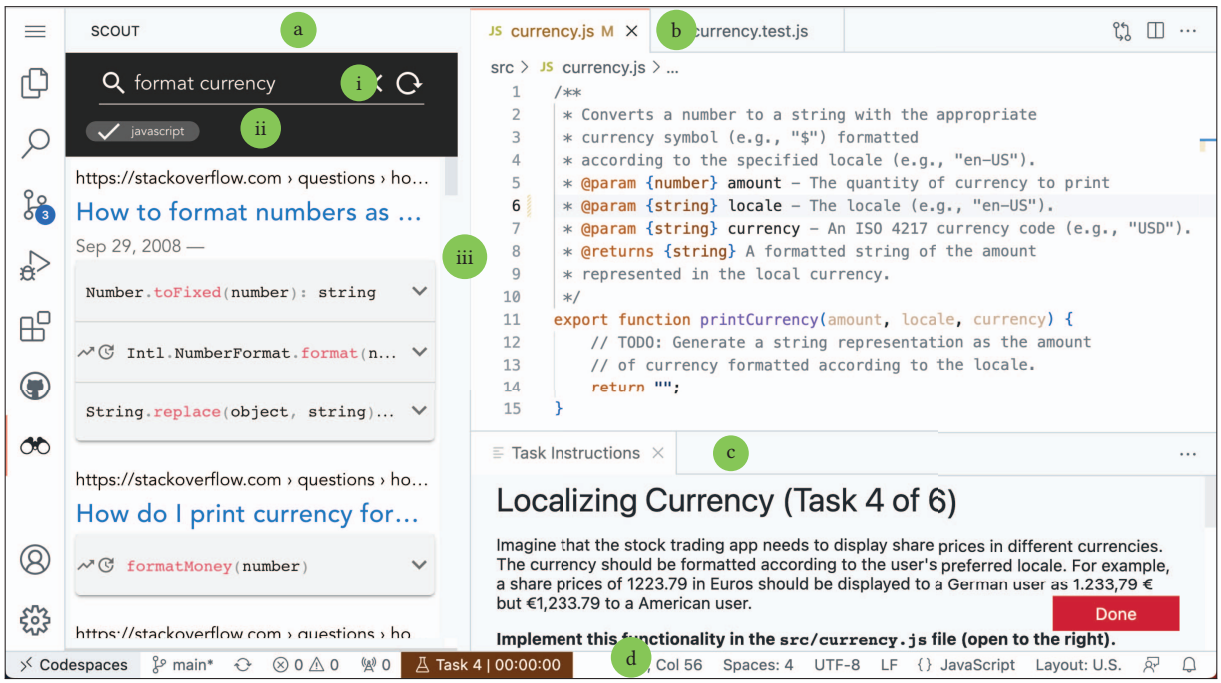


Figure 4: Online study environment. Scout is open in the left panel (a) where a participant has entered a search (i) which includes the selected context terms (ii). Underneath, the search results are summarized as call signatures (iii). The source code and instructions are shown on the right (b)–(c) with the time remaining in the status bar (d).

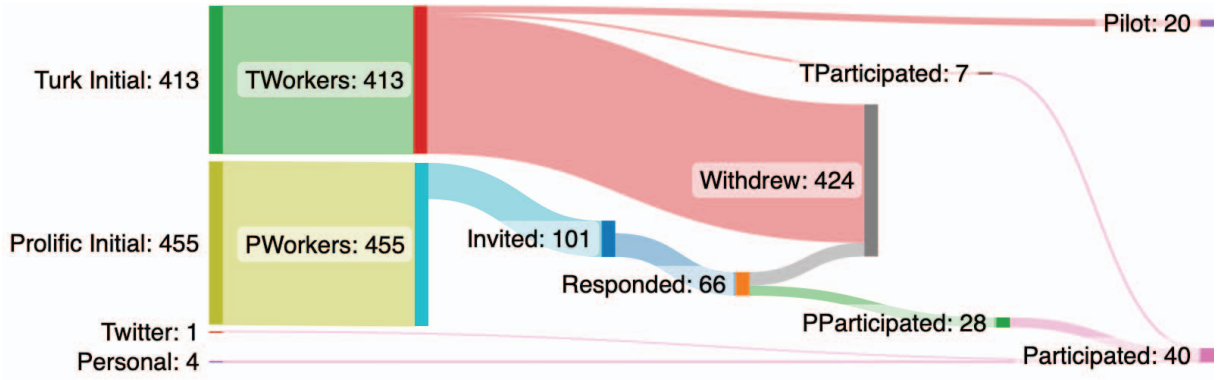


Figure 5: Participant recruitment process.

We recorded participants' interaction with Scout while completing the task including time spent on the task, searches made (incl. context terms), search results (incl. rank, processing time, and signatures), and navigation of the search results (incl. signatures expanded, results opened, and page scrolls). The data was stored in log files committed to participants' provisioned repositories and merged into a single database for analysis along with participants' responses to the post-task questions and the final survey. Our study instruments, data, and analysis scripts are available in our replication package.[6]

[6]https://doi.org/10.17605/OSF.IO/DETRW

4 RESULTS

Scout's main purpose is to direct developers to the API information most relevant to their tasks by reducing the number of posts developers need to investigate. To achieve this objective, Scout uses API call signatures to represent the information in the search results, and context terms from the developer's source code to augment the search terms and rank the signatures. We investigate these aspects in comparison to a traditional Google search approach, focusing on the following research questions:

RQ1 How do API signatures and automatically identified context terms affect developers' search for API information?

RQ2 How does Scouts' API signature representation affect the way developers locate relevant information?
RQ3 How effectively do developers complete tasks with Scout?
RQ4 What is the developers' experience using Scout?

To analyze these questions we use a complementary, mixed methods approach integrating participants' quantitative interaction data and qualitative feedback [5]. When assessing significance, we use Welch's t-test, after verifying normality using the Shapiro-Wilk test, and report normalized effect sizes with Cohen's d.

4.1 User Performance

In this section, we compare participants' searches, the SO posts and answers they viewed, and the success and time for completing the tasks across the 229 cases in the two experimental conditions.

4.1.1 Searching. Overall, participants made significantly fewer searches when using Scout, going down 19% from 1.6 to 1.3 searches on average ($t = -1.98, p = .025; d = 0.26$). In Phase B, searches decreased from 1.3 to 1.1 (15%) and from 2.1 to 1.7 (19%) in Phase C. One explanation for this decrease could be that the API signature summaries allowed participants to recognize relevant solutions faster so they avoided immediately making a new search. Another explanation could be that Scout's automatic addition of context terms to searches led to more suitable initial results requiring participants to make fewer refinements to their search queries.

To investigate the effect of Scouts' context terms, we examined the searches participants made during tasks T5 and T6 (Phase C, Figure 3). We consider the first search participants made during a task as their initial search and any subsequent searches as refinements to improve their results. For each task, we ordered participants' searches by time and manually marked keywords that overlapped with Scout's context terms to account for misspellings and abbreviations (e.g., js for JavaScript). Of the 152 searches participants made in total, 83 were under the control condition (with 39 initial searches, as one participant did not search) and 69 were under Scout (40 initial searches).

Overall, we found that participants refined a smaller proportion of their initial searches under Scout when context terms were provided (14/40 vs 26/39). In the control treatment where no context terms were included, we found that 32/83 searches included a context term that would have been added by Scout automatically. Of these 32 searches, participants manually included a context term in 21 of their 39 initial searches and 11 of the 26 searches they refined. These manually entered terms were distributed across the names of libraries (60%), programming languages (41%), and API calls (2%) aligning with the terms Scout suggests. However, some of the terms Scout suggested did not always align perfectly, with participants removing a library term in 3 of the 69 searches.

When asked how well the suggested context terms aligned with those they would include manually, 28/40 (72%) of participants responded positively. The terms Scout recommended "*seemed appropriate and helpful*" (P15) and were ones they "*would include in order to narrow the search results*" (P24). Participants specifically mentioned they "*liked having the language already embedded*" (P28) and were "*glad [they] didn't have to repeat it each time*" (P23).

> **RQ1 Summary** Scout significantly reduces the number of searches developers perform, with context terms effectively augmenting developers' searches.

4.1.2 Locating. Participants successfully completed 47% (54/115) of the tasks directly from the information provided in Scout's API signature summaries. In contrast, participants only solved 2% (2/114) of the tasks using just the information presented in the Google results (control). The API signature summaries in Scout provided quick access to the relevant information, and participants expanded an average of two signatures and less than one code example. In nearly half of the cases (24/54), participants also copied information directly from the signature summary into their code.

With Scout, participants opened an average of 0.8 SO posts to access further information, significantly fewer than the 2.2 posts they opened when using the Google results ($t = -6.60, p < .001; d = 0.87$). This 64% reduction in navigation cost can partly be attributed to the information and visual cues included in the API signature summaries that made it "*easy to find a solution*" (P1, P2, P11, P17, P24, P26, P36, P39). The check mark and most-upvoted icons helped participants "*decide where to click first*" (P15, P39), while the usage and code tabs specifically made locating a solution easier (P6, P17, P25, P23). Participants also liked "*being able to see function signatures of answers*" (P13), allowing them to compare "*different solutions right in the results page..looking there instead of clicking into every link*" (P25). In contrast, the Google results were not always helpful in locating solutions (P9, P16, P17, P24) since the information might be "*not in context and not necessarily very relevant*" (P4). In these cases, participants instead used the title (P4, P39) to "*click into the SO links and read the thread*" (P9) in the full-page view (P4, P10).

Since there are several answers per post that developers might go through to locate a solution, we also examined the number of answers developers looked at. We considered an answer as viewed when at least half of it was visible in the search interface for at least one second based on scroll events. Overall, study participants viewed an average of 2.5 answers per task when using signature summaries, significantly fewer than the 5.4 answers with Google results ($t = -2.82, p = .002; d = 0.37$). When focusing solely on the times a participant opened a post, the number of answers looked at is almost the same in both conditions, with 3.2 answers read per opened post for tasks with Scout and 3.1 answers with Google (no significant difference, $t = 0.19, p = .577; d = .02$).

When opening a post, Scout enabled participants to jump directly to the answer corresponding to a specific API signature via a link, rather than examining each answer starting from the top of a post. Participants used this link for 8 of the 61 (13%) tasks where they opened a post, viewing an average of 1.8 answers once the post was open, including the linked answer.

> **RQ2 Summary** API signature summaries make relevant information directly accessible so that developers open significantly fewer SO posts and look through significantly fewer answers.

4.1.3 Task Completion. Overall, we found that there was no significant difference in participants' success rate or completion times between the two conditions. In terms of success rate, we marked a task as successful if the participant's solution passed all of the provided unit tests. While participants were slightly more successful when using API signatures, completing 90/115 tasks compared to the 85/114 tasks when using Google results, the difference was not significant (Two-Proportions Z-Test $\chi^2 = 0.2537, df = 1, p = .307$). There was also no significant difference when excluding cases that exceeded the allotted time.

In cases where participants were unsuccessful, we coded the feedback they provided to understand their main challenges under the two conditions. When using Google results (without context terms), participants' main issue was "*finding the right search terms*" (P14; P10, P17, P20, P24, P27). When using Scout's API signatures, "*finding a relevant code sample seemed straightforward*" but participants wasted time resolving variable name typos (P1, P26) and not using the signatures sooner (P25; P33). Despite their similar performance under the two treatments, 36/40 participants indicated a strong preference for API signatures, as P39's feedback illustrates: "*[Google] was missing the organized function [API] call signatures like the previous tasks. I liked those!*".

Regarding completion time, participants were generally able to complete the tasks in the allotted time. We measured participants' completion times from the start of the task to when they clicked done, including the time they spent comprehending the task and implementing a solution (Figure 6). While tasks were overall completed slightly faster on average with the control condition (4m18s) than with Scout (4m24s), a two-way repeated measures ANOVA, treating the task as a blocking factor, showed that the treatment had no significant effect on participants' completion times ($F(1, 227) = 0.06, p = .81$). Note that we are missing the completion time for one participant for T6 due to technical issues. In 67 cases, participants exceeded the allotted time, including four large outliers in T1 (19m6s), T2 (27m22s), T3 (17m32s), and T6 (13m32s), with 34 cases in the control condition and 33 with Scout. Ultimately, 6 of the 34 cases in the control condition were successfully completed and 8 of the 33 using Scout. Our results remained consistent when excluding these cases: participants took 3m0s on average for the control condition and 3m6s for Scout, and the ANOVA showed no significant difference.

> **RQ3 Summary** Despite the significant reduction in information when using API signature summaries, there is no significant effect on developers' task success or completion time between conditions.

4.2 Tool Performance

It is important that search tools respond quickly with useful suggestions that help developers successfully complete their tasks. Here we report on Scout's signature ranking and processing latency.

4.2.1 Conciseness and Ranking. Overall, Scout provided participants with succinct representations of the information available in their search results by surfacing task-relevant API signatures. The SO posts in the search results for the 69 searches participants made

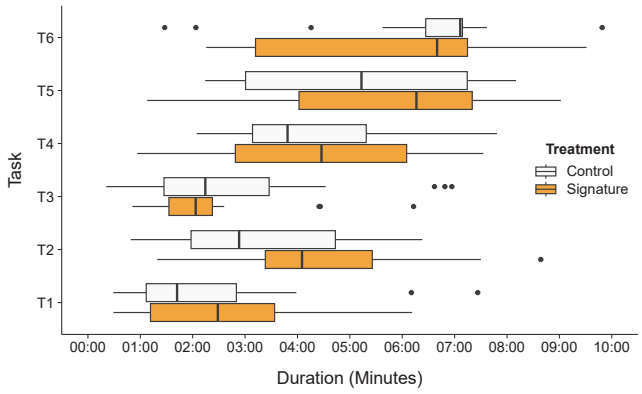


Figure 6: Distribution of task completion times. For readability, we do not show the four large outliers.

using Scout in T5 and T6 contained an average of 30k words and 292 LoC with 122 signatures across 26 answers. Instead of participants manually opening posts and looking through the answers for the appropriate API information, Scout provided a concise overview, summarizing the results into 23 signatures using 86 words and 23 LoC on average.

To examine how effectively Scout ranks API signatures within each SO post, we manually compared participants' solutions with the API signatures shown by Scout for the last search made by each participant during T5 and T6. For these searches, there were 15 signatures on average in each post from which Scout selected the 3 most relevant considering the participant's code. We found that 31% of participants' solutions used the first signature, 64% used one of the first 3 signatures, and 97% used one of the first 10. For some tasks, participants were impressed with the signature ranking noting that "*the first item..had the answer I needed*" (P32; P20). However, the ranking was not always perfect (P9, P28, P29, P31, P35) leading participants to examine signatures that were not always relevant (P5, P10, P32). In these cases, signatures can help mitigate the impact of ranking by enabling developers to directly assess the relevance of the results without opening the full SO post: "*It was really easy to find the answer, even though it wasn't the first one because [the signatures] were nicely collapsed. The example also helped*" (P31). Participants also suggested that the ranking could be improved by incorporating the number of upvotes (P27) and answer recency (P8, P24) into the ranking algorithm, and prioritizing signatures with shorter code examples (P26, P32), or those meeting technical requirements such as the language revision (P20).

4.2.2 Latency. Scout creates the API signature summaries by processing the first 10 SO posts returned within Google's search results. Since this processing happens in real time and can impact Scout's usability, we analyzed the latency it adds to the Google search results. We found that Scout took an average of 4.4s to display the first search result and an average of 5.0s for Google's top-ranked result, on the 8-core virtual machines used during the study. While this latency is below the 10s tolerable wait time for web-based information retrieval tasks, it is longer than the sub-2s ideal wait time [27]. According to participants, the latency was not a major factor

affecting Scout's usability and is partly offset by reducing navigational overhead (Section 4.1.2), which may result in an overall faster "*experience*" (P15). However, some participants did mention that the "*search could be faster*" compared to Google's highly-optimized search engine (P5, P8, P11, P18).

4.3 User Experience

For developers to actually find solutions using a search tool, it must be usable and support the types of searches developers make during their tasks. We examine these aspects below.

4.3.1 Usability. Participants indicated that Scout has *good* usability, assigning it an average system usability score of 78 (a score of 70 is average) [3]. They found it "*helpful to have [the results] in the IDE*" (P39) so they did not "*have to leave the [..] code*" (P15) and "*loved that [they] didn't have to switch tabs/windows*" (P19). However, one usability challenge was the limited space available in the IDE (P4, P6, P8, P11, P16, P18, P23, P29, P37) which made the interface "*a bit too busy*" (P28) and "*slightly harder to navigate*" (P13).

When comparing the API signature summaries with standard Google result summaries, participants generally found that signatures were quicker to scan for useful solutions saving them time and mental energy processing full SO posts. However, participants noted that with signatures it could be "*a little tough to know which result to use [from] the initial view*" (P15) without more documentation (P5, P6, P13, P23). In contrast, the textual content of standard Google results resized to fit the space available in the IDE but required additional navigation to "*dig*" through code, comments, and links to get an actual solution (P20, P23).

4.3.2 Search Suitability. Participants indicated that signature summaries are best suited to searches involving *basic* tasks where they are familiar with the overall approach and need to *recall* details about *common* API calls such as syntax using *code examples*. However, they are not as useful for searches involving *specific* or *complex* tasks that require a *detailed understanding* of possible solutions using *multi-line code examples*, or for tasks that are *conceptual* in nature. Participants reported that 58% of their day-to-day searches are for basic tasks while 38% are for complex tasks. To better support these two types of searches, participants suggested "*having both [summaries] at once*" (P15, P31) by adding full API signatures or even "*just function names*" (P20, P23) to the Google results.

4.3.3 IDE Integration. Participants were generally positive about Scout's integration into the IDE, but also suggested some ideas to further the integration. Concretely, participants suggested adding a "*copy*" button (P28, P33) that "*drops the [summary] code into the source file*" (23), and altering the summary code to match the source code; e.g., by "*matching the parameter [names]*" (P31) and "*highlighting variables*" (P16). To maximize the limited window space, participants suggested using "*[line wrapping] for signature summaries to reduce horizontal scrolling*" (P24), and providing "*tooltips to view text snippets related to code*" (P4, P24) with an option to "*expand the Scout window or open the links in new windows*" (P18).

RQ4 Summary Participants valued Scout's concise API signature presentation and the tight IDE integration that speeds up scanning and locating solutions. Scout is particularly suitable for basic search tasks comprising more than half of developers' day-to-day work.

5 DISCUSSION

In this section we discuss some challenges and solutions for extending Scout to developers' workflows in practice.

5.1 Contextualizing Developers' Searches

Constructing effective queries can be difficult, leading developers to invest time and mental effort examining results which are not relevant to their task. We found that many of the terms developers include in their queries to filter the search results can be automatically extracted from their source code and, by presenting the terms explicitly, enable developers to actively guide the search process. Participants found the terms that Scout included in their searches aligned with their expectations and even thought that additional terms, such as variable types, and terms from specially formatted comments like `JSDoc` (P4, P26, P28, P37), would be helpful. However, they acknowledged that it would not always "*reasonable [..] when [the terms] don't show up in code*" (P31). Some participants were concerned that Scout might "*infer too [many terms]*" (P15) from longer functions that could introduce noise into their searches (P8). Future approaches could automatically include the 2–3 terms closest to the developer's cursor and offer remaining terms through autocomplete. Participants also thought that providing "*suggestions like Google's 'Did you mean...?'*" (P19) could help them "*search for similar problems*" (P2).

5.2 Alternative API Oracles

The focus of our experiment was to study the effect of using API signatures to orient developers within their search results and help them locate a solution more directly, independent of where the API signature information was derived from. While there are existing approaches that share similar goals, such as Prompter [33], PostFinder [37], and Biker [17], they focus on improving the underlying search results through custom search engines. Instead, Scout focuses on aligning the presentation of the information contained within a set of search results to the developer's task. By decoupling the presentation of search results, Scout works on top of any search engine providing SO results. Due to its ubiquity, quality of results, and ease of use, we chose to use Google as our baseline search provider within Scout.

Tools based on LLMs, such as ChatGPT and Copilot, are another source of API information but, similar to web search, require developers to either craft effective queries or look through the responses to identify relevant information. In the case of ChatGPT's conversational interface, developers often have to try multiple queries to obtain a complete solution for their task [28]. This process adds overhead as developers have to read the frequently verbose, and structurally variable, responses and think about how to refine their queries to elicit the desired information [18]. Copilot avoids queries

by generating multiple code solutions from cues in the developer's source code. However, the code-only representation of solutions makes them difficult for developers to fully assess, which is important since the solutions can be wrong in subtle ways [30]. Further, the size of the solutions, and Copilot's autocomplete interface, which provides one solution at a time, can make it hard for developers to efficiently compare alternative solutions. The presentation system used by Scout can be applied to existing LLM tools to enable their recommendations to be presented in a way that is better tailored to the developer's task.

5.3 Summarizing Task-Centric Results

Text snippets, like the ones typically used by web search engines, can be generated robustly but have inherent limitations when summarizing developer-specific results such as being "*out of context*" (P4), containing "*unnecessary details*" (P17), and preventing developers from "*easily seeing and comparing answers*" (P15). For searches about APIs, we found that signatures address these limitations by enabling developers to locate solutions for their tasks more directly. To make the summaries *easier to navigate*, participants suggested including the age (P10), number of votes, and source answers for each signature (P13, P24), and always providing the full signature "*showing what parameters the function accepts, rather than just based on its usage in that specific answer*" (P38). Participants also suggested removing the Usage tab (P23), or defaulting to the Code tab (P29) to "*save an extra click*" (P19). For code examples, participants wanted "*longer documentation*" or a "*small summary*" (P5, P36) with "*comments in the code*" (P39, P7) and links to the function's official documentation (P10, P13, P36).

While signatures can help developers save time by directly surfacing the API information contained in their search results, the latency from processing the search results in real time can diminish Scout's overall time savings. The primary factors contributing to Scout's latency are fetching the posts from SO and processing the code examples into ASTs. In future work we will examine approaches to minimize real time processing latency, for example, by requesting posts directly through the SO API rather than using proxied network requests, and by caching the code example ASTs.

5.4 Completing Tasks

We designed Scout to reduce cognitive load and help developers focus on their tasks. However, it is difficult to reliably measure these aspects directly so we instead examined completion times and success rates, which are common measures when comparing alternative tool designs. We did not find significant differences in completion time or success rate between the Google and Scout conditions. On one hand, we view this as a positive result since we compared Scout's novel interface, which participants had to learn while completing actual development tasks, with Google, one of the most commonly used web search tool by developers. On the other hand, we expect Scout could be faster in practice by enabling developers to search directly in the IDE, avoiding the cognitive overhead associated with the context switches and window management that occur when using a separate browser application. For this study, we compared both conditions within the IDE to avoid any confounds and privacy concerns outside of our main evaluation of Scout's API signatures.

5.5 Threats to Validity

We describe the limitations of our study below.

Internal Validity. This study uses a within-subject design which introduces confounds such as carryover and ordering effects. We randomly assigned participants to one of four trial groups, following a balanced Latin Square design, to help mitigate these effects, and included breaks to limit participant fatigue.

Before starting the study, we conducted a power analysis to estimate the number of participants necessary to observe statistically significant results. While we used the recommended effect size of $d = 0.5$, our post-hoc power analysis indicates the actual effect size was much smaller at $d = 0.13$ limiting our interpretation of success rate and completion time. By using many shorter tasks, participants' overall completion times may have been dominated by aspects of the task other than searching, including reading instructions, comprehending code, and testing solutions.

External Validity. The results from this study may not generalize to all developers and tasks. We chose to run the study primarily on paid platforms to recruit a large number of diverse participants who would have been difficult to contact directly. However, by conducting this study on paid platforms, participants may have been incentivized to misrepresent their ability or otherwise complete the study differently than expected (e.g., by working on multiple studies simultaneously). We used a validated questionnaire to prescreen participants, and required participants to successfully complete the tutorial task and attempt all of the tasks before being compensated.

The tasks we used for the study were designed to represent individual components of larger features found in real software systems. We chose to use short tasks, presented in isolation, to provide more comparison points and to help ensure participants could make progress while keeping the overall study duration reasonable. We intended this design to match how developers decompose their tasks and search for solutions in practice.

6 RELATED WORK

Developers perform two steps to locate solutions on the web: first, they generate search terms and then they navigate among the result pages. Prior work has generally approached these steps independently by creating tools that help developers either construct search queries using code terms in the IDE or navigate the result pages by transforming their content using different summarization techniques in the web browser.

6.1 Constructing Search Queries

Coming up with effective search terms is difficult and often requires developers to refine their terms after investigating the results, which can take considerable time [35, 50]. One way to minimize this refinement phase is to use keywords from the text in the IDE, such as source code and error messages, to generate queries automatically. An early example of this approach is `Strathcona`, which automatically recommends API usage examples from an oracle based on the structural context of the source code selected by a developer in their

IDE [15]. Pycee and Maestro use stack traces to generate searches to help developers locate information about program errors [23, 43]. Fishtail, Reverb, and Prompter use the code entities developers have recently interacted with to recommend previously viewed web pages ([40, 41]) and relevant SO posts ([33]). StackInTheFlow, PostFinder, FaCoY, and Bing Developer Assistant generate searches on-demand from the code selected by developers in the IDE [13, 20, 37, 49].

However, these approaches rely on custom search indexes that may not always have the most up-to-date information (e.g., SO data dumps). It can also be difficult for developers to guide these approaches using their knowledge of the task since the search queries are generated and run transparently. Instead, Scout offers code terms that developers can use to augment their queries, similar to Blueprint [7], allowing them to actively control the results from standard search engines with minimal effort.

6.2 Transforming Search Results

Generating effective queries is only the first part of locating solutions on the web: developers still need to find the specific kinds of information relevant to their task from their search results, such as explanations and examples of API usage [16]. Unfortunately, finding APIs within the search results can be difficult when developers are only given limited information about the result such as the title (e.g., Google, Prompter, PostFinder).

One way to help developers effectively navigate the results is to annotate them with developer-specific information. Mica and Assieme extract the function and type names, respectively, from search result pages so that developers can easily assess and refine the results [14, 42]. Libra adds a plot to Google's search result page visualizing results according to the number and diversity of overlapping code terms, helping guide developers toward pages with more details or alternative solutions [34]. A tool by Liu et al. identifies latent developer needs within SO questions to help developers identify posts relevant to their task [22].

Alternatively, approaches can provide a summary of the results so developers do not need to examine each result individually. Example Overflow presents the top 5 accepted code examples from SO search results in a single view allowing developers to easily compare them [48]. While such code examples can be helpful, they can be time-consuming to read and may not reflect common usage. Exempla Gratis identifies common usage patterns across multiple code examples to identify the idiomatic usage of developer-specified API methods [4]. Instead of code examples, AnswerBot focuses on textual content to extract a diverse summary across answers to give developers an overview of the entire SO post [46].

Some of these approaches have focused on summarizing API information. CAPS summaries SO posts describing API issues to help developers improve the API design [2], while a tool by Campos et al. identifies highly-voted answers across SO posts demonstrating how to use an API [8]. Opiner provides sentiment about Java packages found in SO code examples based on the answer text to help developers decide between alternative APIs. Biker uses heuristics to extract API entities from hyperlinks and code tags in the top-50 SO posts related to a developer's search by comparing them to a curated oracle of Java API names. The entities are mapped to their fully qualified name and ranked by frequency and similarity of the post title with the search terms. The API description, a list of similar questions, and code examples are presented in a dedicated browser window using plain text [17].

While we share a similar objective with Biker, our approach differs in several key ways. We focus on presenting APIs interactively in the IDE where developers can compare different options before expanding detailed usage examples. Our approach works in real time using Google search results avoiding the need to construct and update a custom search engine. Finally, we consider where the extracted APIs will be used in the developer's code when making recommendations.

7 CONCLUSION

In this paper we introduced Scout, a tool to help developers locate API information within their search results by extracting, ranking, and presenting compact API signatures directly within the IDE. Scout works on top of existing search engines by incorporating terms found within the developer's source code to augment their search queries and adjust the search results. In a controlled experiment with 40 developers, we found that presenting search results as API signatures enabled developers to easily compare different APIs and interactively drill down to details when needed, significantly reducing the amount of information they examined. Participants were able to complete API search tasks with Scout's novel interface while maintaining similar task completion times and success rates.

Our findings indicate that the way information about source code is presented can affect the amount of effort required for developers to complete their tasks. Code context provides an effective mechanism for tailoring the representation of developer's information resources to their task. In the case of API search tasks, an API signature representation helps developers assess the relevance of search result pages and, for many searches, provides exactly the required information.

REFERENCES

[1] Rabe Abdalkareem, Emad Shihab, and Juergen Rilling. 2017. What Do Developers Use the Crowd For? A Study Using Stack Overflow. *IEEE Software* 34, 2 (March 2017), 53–60.

[2] Md Ahasanuzzaman, Muhammad Asaduzzaman, Chanchal K. Roy, and Kevin A. Schneider. 2020. CAPS: A Supervised Technique for Classifying Stack Overflow Posts Concerning API Issues. *Empirical Software Engineering (EMSE)* 25, 2 (March 2020), 1493–1532.

[3] Aaron Bangor, Philip Kortum, and James Miller. 2009. Determining What Individual SUS Scores Mean: Adding an Adjective Rating Scale. *Journal of Usability Studies* 4, 3 (May 2009), 114–123.

[4] Celeste Barnaby, Koushik Sen, Tianyi Zhang, Elena Glassman, and Satish Chandra. 2020. Exempla Gratis (E.G.): Code Examples for Free. In *Proceedings of the Joint Meeting on European Software Engineering Conference and the Symposium on the Foundations of Software Engineering (ESEC/FSE)*. 1353–1364.

[5] Patricia Bazeley. 2018. Complementary Analysis of Varied Data Sources. In *Integrating Analyses in Mixed Methods Research*. 91–125.

[6] Nick C. Bradley, Thomas Fritz, and Reid Holmes. 2022. Sources of Software Development Task Friction. *Empirical Software Engineering (EMSE)* 27, 7 (Sept. 2022), 34 pages.

[7] Joel Brandt, Mira Dontcheva, Marcos Weskamp, and Scott R. Klemmer. 2010. Example-Centric Programming: Integrating Web Search into the Development Environment. In *Proceedings of the Conference on Human Factors in Computing Systems (CHI)*. 513–522.

[8] Eduardo C. Campos, Lucas B. L. de Souza, and Marcelo de A. Maia. 2016. Searching Crowd Knowledge to Recommend Solutions for API Usage Tasks. *Journal of Software: Evolution and Process* 28, 10 (July 2016), 863–892.

[9] Joseph Chee Chang, Nathan Hahn, Yongsung Kim, Julina Coupland, Bradley Breneisen, Hannah S Kim, John Hwong, and Aniket Kittur. 2021. When the Tab

Comes Due:Challenges in the Cost Structure of Browser Tab Usage. In *Proceedings of the Conference on Human Factors in Computing Systems (CHI)*. 1–15.

[10] Anastasia Danilova, Alena Naiakshina, Stefan Horstmann, and Matthew Smith. 2021. Do You Really Code? Designing and Evaluating Screening Questions for Online Surveys with Programmers. In *Proceedings of the International Conference on Software Engineering (ICSE)*. 537–548.

[11] Tore Dybå, Vigdis By Kampenes, and Dag I. K. Sjøberg. 2006. A Systematic Review of Statistical Power in Software Engineering Experiments. *Information and Software Technology* 48, 8 (2006), 745–755.

[12] G. Gao, F. Voichick, M. Ichinco, and C. Kelleher. 2020. Exploring Programmers API Learning Processes: Collecting Web Resources as External Memory. In *Proceedings of the Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. 1–10.

[13] Chase Greco, Tyler Haden, and Kostadin Damevski. 2018. StackInTheFlow: Behavior-Driven Recommendation System for Stack Overflow Posts. In *Proceedings of the International Conference on Software Engineering (ICSE)*. 5–8.

[14] Raphael Hoffmann, James Fogarty, and Daniel S. Weld. 2007. Assieme: Finding and Leveraging Implicit References in a Web Search Interface for Programmers. In *Proceedings of the Symposium on User Interface Software and Technology (UIST)*. 13–22.

[15] Reid Holmes and Gail C. Murphy. 2005. Using structural context to recommend source code examples. In *Proceedings of the International Conference on Software Engineering (ICSE)*. 117–125.

[16] Andre Hora. 2021. Googling for Software Development: What Developers Search For and What They Find. In *Proceedings of the International Conference on Mining Software Repositories (MSR)*. 317–328.

[17] Qiao Huang, Xin Xia, Zhenchang Xing, David Lo, and Xinyu Wang. 2018. API Method Recommendation without Worrying about the Task-API Knowledge Gap. In *Proceedings of the International Conference on Automated Software Engineering (ASE) (ASE 2018)*. 293–304.

[18] Samia Kabir, David N. Udo-Imeh, Bonan Kou, and Tianyi Zhang. 2023. *Who Answers It Better? An In-Depth Analysis of ChatGPT and Stack Overflow Answers to Software Engineering Questions.* arXiv:2308.02312

[19] Katja Kevic and Thomas Fritz. 2014. Automatic Search Term Identification for Change Tasks. In *Proceedings of the International Conference on Software Engineering (ICSE)*. 468–471.

[20] Kisub Kim, Dongsun Kim, Tegawendé F. Bissyandé, Eunjong Choi, Li Li, Jacques Klein, and Yves Le Traon. 2018. FaCoY - A Code-to-Code Search Engine. In *Proceedings of the International Conference on Software Engineering (ICSE)*. 946–957.

[21] Jiakun Liu, Sebastian Baltes, Christoph Treude, David Lo, Yun Zhang, and Xin Xia. 2021. Characterizing Search Activities on Stack Overflow. In *Proceedings of the Joint Meeting on European Software Engineering Conference and the Symposium on the Foundations of Software Engineering (ESEC/FSE)*. 919–931.

[22] Mingwei Liu, Xin Peng, Andrian Marcus, Shuangshuang Xing, Christoph Treude, and Chengyuan Zhao. 2021. API-Related Developer Information Needs in Stack Overflow. *Transactions on Software Engineering (TSE)* 48, 11 (Nov. 2021), 4485–4500.

[23] Sonal Mahajan, Negarsadat Abolhassani, and Mukul R. Prasad. 2020. Recommending Stack Overflow Posts for Fixing Runtime Exceptions Using Failure Scenario Matching. In *Proceedings of the Joint Meeting on European Software Engineering Conference and the Symposium on the Foundations of Software Engineering (ESEC/FSE)*. 1052–1064.

[24] Collin McMillan, Mark Grechanik, Denys Poshyvanyk, Qing Xie, and Chen Fu. 2011. Portfolio: Finding Relevant Functions and Their Usage. In *Proceedings of the International Conference on Software Engineering (ICSE)*. 111–120.

[25] Roberto Minelli, Andrea Mocci, and Michele Lanza. 2015. The Plague Doctor: A Promising Cure for the Window Plague. In *Proceedings of the International Conference on Software Maintenance and Evolution (ICSME)*. 182–185.

[26] Tamara Munzner. 2014. *Visualization Analysis and Design.* AK Peters/CRC press, Chapter 14.

[27] Fiona Fui-Hoon Nah. 2004. A Study on Tolerable Waiting Time: How Long Are Web Users Willing to Wait? *Behaviour & Information Technology* 23, 3 (May 2004), 153–163.

[28] Daye Nam, Andrew Macvean, Vincent Hellendoorn, Bogdan Vasilescu, and Brad Myers. 2023. *In-IDE Generation-based Information Support with a Large Language Model.* arXiv:2307.08177

[29] Seyed Mehdi Nasehi, Jonathan Sillito, Frank Maurer, and Chris Burns. 2012. What Makes a Good Code Example?: A Study of Programming Q&A in StackOverflow. In *Proceedings of the International Conference on Software Maintenance (ICSM)*. 25–34.

[30] Nhan Nguyen and Sarah Nadi. 2022. An empirical evaluation of GitHub copilot's code suggestions. In *Proceedings of the International Conference on Mining Software Repositories (MSR)*. 1–5.

[31] Cole S. Peterson, Jonathan A. Saddler, Natalie M. Halavick, and Bonita Sharif. 2019. A Gaze-Based Exploratory Study on the Information Seeking Behavior of Developers on Stack Overflow. In *Proceedings of the Conference on Human Factors in Computing Systems (CHI)*. 1–6.

[32] Jan Pilzer, Raphael Rosenast, André N. Meyer, Elaine M. Huang, and Thomas Fritz. 2020. Supporting Software Developers' Focused Work on Window-Based Desktops. In *Proceedings of the Conference on Human Factors in Computing Systems (CHI)*. 1–13.

[33] Luca Ponzanelli, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, and Michele Lanza. 2014. Mining StackOverflow to Turn the IDE into a Self-confident Programming Prompter. In *Proceedings of the International Conference on Mining Software Repositories (MSR)*. 102–111.

[34] Luca Ponzanelli, Simone Scalabrino, Gabriele Bavota, Andrea Mocci, Rocco Oliveto, Massimiliano Di Penta, and Michele Lanza. 2017. Supporting Software Developers with a Holistic Recommender System. In *Proceedings of the International Conference on Software Engineering (ICSE)*. 94–105.

[35] Md Masudur Rahman, Jed Barson, Sydney Paul, Joshua Kayani, Federico Andrés Lois, Sebastián Fernandez Quezada, Christopher Parnin, Kathryn T. Stolee, and Baishakhi Ray. 2018. Evaluating How Developers Use General-Purpose Web-Search for Code Retrieval. In *Proceedings of the International Conference on Mining Software Repositories (MSR)*. 465–475.

[36] Mohammad M. Rahman, Chanchal K. Roy, and David Lo. 2019. Automatic Query Reformulation for Code Search Using Crowdsourced Knowledge. *Empirical Software Engineering (EMSE)* 24, 4 (Aug. 2019), 1869–1924.

[37] Riccardo Rubei, Claudio Di Sipio, Phuong T. Nguyen, Juri Di Rocco, and Davide Di Ruscio. 2020. PostFinder: Mining Stack Overflow Posts to Support Software Developers. *Information and Software Technology* 127 (Nov. 2020), 16 pages.

[38] Daniel Russo. 2022. Recruiting Software Engineers on Prolific. In *International Workshop on Recruiting Participants for Empirical Software Engineering*. 2 pages.

[39] Caitlin Sadowski, Kathryn T. Stolee, and Sebastian Elbaum. 2015. How Developers Search for Code: A Case Study. In *Proceedings of the International Symposium on Foundations of Software Engineering (FSE)*. 191–201.

[40] Nicholas Sawadsky and Gail C. Murphy. 2011. Fishtail: From Task Context to Source Code Examples. In *Proceedings of the Workshop on Developing Tools as Plug-ins*. 48–51.

[41] Nicholas Sawadsky, Gail C. Murphy, and Rahul Jiresal. 2013. Reverb: Recommending Code-related Web Pages. In *Proceedings of the International Conference on Software Engineering (ICSE)*. 812–821.

[42] J. Stylos and B.A. Myers. 2006. Mica: A Web-Search Tool for Finding API Components and Examples. In *Proceedings of the Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. 195–202.

[43] E. Thiselton and C. Treude. 2019. Enhancing Python Compiler Error Messages via Stack. In *Proceedings of the International Symposium on Empirical Software Engineering and Measurement (ESEM)*. 1–12.

[44] Ying Wang, Bihuan Chen, Kaifeng Huang, Bowen Shi, Congying Xu, Xin Peng, Yijian Wu, and Yang Liu. 2020. An Empirical Study of Usages, Updates and Risks of Third-Party Libraries in Java Projects. In *Proceedings of the International Conference on Software Maintenance and Evolution (ICSME)*. 35–45.

[45] Xin Xia, Lingfeng Bao, David Lo, Pavneet Singh Kochhar, Ahmed E. Hassan, and Zhenchang Xing. 2017. What Do Developers Search for on the Web? *Empirical Software Engineering (EMSE)* 22, 6 (Dec. 2017), 3149–3185.

[46] Bowen Xu, Zhenchang Xing, Xin Xia, and David Lo. 2017. AnswerBot: Automated Generation of Answer Summary to Developers' Technical Questions. In *Proceedings of the International Conference on Automated Software Engineering (ASE)*. 706–716.

[47] Annie T. T. Ying and Martin P. Robillard. 2014. Selection and Presentation Practices for Code Example Summarization. In *Proceedings of the International Symposium on Foundations of Software Engineering (FSE)*. 460–471.

[48] Alexey Zagalsky, Ohad Barzilay, and Amiram Yehudai. 2012. Example Overflow: Using Social Media for Code Recommendation. In *Proceedings of the International Workshop on Recommendation Systems for Software Engineering (RSSE)*. 38–42.

[49] Hongyu Zhang, Anuj Jain, Gaurav Khandelwal, Chandrashekhar Kaushik, Scott Ge, and Wenxiang Hu. 2016. Bing Developer Assistant: Improving Developer Productivity by Recommending Sample Code. In *Proceedings of the International Symposium on Foundations of Software Engineering (FSE)*. 956–961.

[50] Neng Zhang, Qiao Huang, Xin Xia, Ying Zou, David Lo, and Zhenchang Xing. 2022. Chatbot4QR: Interactive Query Refinement for Technical Question Retrieval. *Transactions on Software Engineering (TSE)* 48, 4 (April 2022), 1185–1211.