Laminar 2.0: Serverless Stream Processing with Enhanced Code Search and Recommendations

Daniel Rotchford
University of St Andrews
School of Computer Science
dr207@st-andrews.ac.uk

Samuel Evans
University of St Andrews
School of Computer Science
sje7@st-andrews.ac.uk

Rosa Filgueira
University of Edinburgh
EPCC
r.filgueira@epcc.ed.ac.uk

***Abstract*—This paper presents `Laminar 2.0`, an enhanced serverless framework for running `dispel4py` streaming workflows. Building on `Laminar 1.0`, this version introduces improved dependency management, client-server functionality, and advanced deep learning models for semantic search. Key innovations include a structural *code-to-code* search using simplified parse syntax trees (SPTs) for detecting similar Processing Elements (PEs) or workflows, even from incomplete code. Additionally, `Laminar 2.0` optimizes *text-to-code* search through better preprocessing of PEs. Our evaluation shows significant performance improvements over the previous version.**

***Index Terms*—Serverless computing, streaming workflows, semantic code search, Laminar, dispel4py.**

I. Introduction

Serverless computing [Kumar(2019)] has emerged as a transformative paradigm in cloud computing, offering scalability, cost-effectiveness, and simplicity in deploying applications. However, the surge in data-intensive applications and the demand for real-time processing present new challenges [Shafiei et al.(2022)] for existing frameworks. Traditional serverless architectures struggle to handle continuous data streams efficiently, resulting in bottlenecks and latency issues. Supporting stateful computations within a serverless environment also becomes complex due to the need to manage state across distributed, ephemeral instances.

To address these challenges, we introduced `Laminar 1.0` [Zahra et al.(2023)], an open-source serverless stream-based processing framework with deep learning code search. Unlike traditional frameworks, `Laminar` effectively handles data streams and supports stateful computations by leveraging the `dispel4py` Python library [Filgueira et al.(2014)], [Liang et al.(2023)]. `dispel4py`'s support for parallelism enables concurrent data processing, while abstract workflow descriptions in Python empower users to construct complex stream processing pipelines.

Building on the success of `Laminar 1.0`, we present `Laminar 2.0` [1], which introduces significant enhancements, including advanced deep learning-based semantic code search, code completion, and code summarization capabilities. A critical aspect of optimizing workflow development and execution in serverless environments is the effectiveness of search capabilities within registries. These searches can be categorized into literal and semantic searches, with semantic searches further divided into *text-to-code* and *code-to-code* searches.

In `Laminar 1.0`, we utilized the `UniXcoder` model [Guo et al.(2022)] for *text-to-code* searches and the `ReACC-py-retriever` [Lu et al.(2022)] for *code-to-code* PE searches. While effective, these models had limitations with partial and structurally diverse code snippets. To address these limitations and enhance search capabilities, we have integrated the structural code search approach proposed in `Aroma` [Luan et al.(2019)], originally designed for Java code snippets. This method uses simplified parse trees to compare code snippets based on their structure, enabling more accurate *code-to-code* searches, especially for incomplete code fragments. Integrating `Aroma` into `Laminar 2.0` significantly enhances code recommendations and search functionalities. The main contributions of this work are:

- Enhanced client-side functionality with improved usability and dynamic workflow execution.
- Full Python 3.10+ compatibility, leveraging the latest features for better performance.
- Support for dynamic process allocationand real-time data streams within serverless environments.
- Streamlined workflow registration, resource management, and auto-provisioning.
- Optimized execution engine with Dockerized architecture for scalable deployment.
- Advanced search and code recommendation capabilities, including structural and semantic searches.
- Improved automated description generation for PEs and workflows, boosting search accuracy.

The paper is organized as follows: Section II reviews relevant technologies. Section III offers an overview of `Laminar 2.0`, followed by key enhancements in Section IV . Section V explores advanced search functionalities. Section VI details the code recommendation including the integration of `Aroma`. Section VII presents performance evaluations, and Section VIII compares `Laminar 2.0` with other frameworks. Finally, Section IX concludes the paper and suggests future work.

[1] https://github.com/StreamingFlow

979-8-3503-5554-3/24/$31.00 ©2024 IEEE
DOI 10.1109/SCW63240.2024.00261

II. Background

A. dispel4py

`dispel4py` [2] is a parallel stream-based dataflow framework for data-intensive applications. It simplifies workflow creation and execution through automatic parallelization and abstract workflow descriptions. Workflows are directed acyclic graphs (DAGs) with nodes representing Processing Elements (PEs) and edges representing data flow, enabling efficient, concurrent data processing. Key components include:

- **Processing Elements (PEs):** Fundamental units of computation that perform specific tasks and can be reused across different workflows.
- **Abstract Workflow**: Represents logical connections between PEs, outlining computational sequences and data transformations. It is what the user describes.
- **Mappings:** Translates abstract workflows onto execution systems, including sequential and parallel (e.g., *MPI* [Forum(1994)], *Multiprocessing* [3], *Redis [Eddelbuettel(2022)]*) alternatives.
- **Concrete Workflow**: During enactment, `dispel4py` builds the concrete workflow based on user-specified mappings and process numbers. This workflow is executed by the compute infrastructure.
- **Workload Allocation**: `dispel4py` supports static workload distribution with *mpi* and *multiprocessing* mappings. Dynamic allocation, introduced in [Liang et al.(2022)], allows adaptive resource allocation to PEs using the *Redis* mapping.

Figure 1 illustrates the `dispel4py` workflow architecture, showing interconnected PEs within a workflow graph executed in parallel. Users define abstract workflow graphs, specify mappings and process numbers, and `dispel4py` automatically creates and executes the concrete workflow.

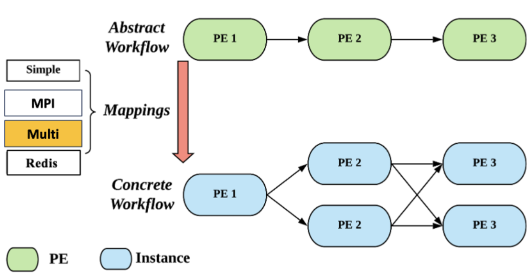


Fig. 1: Example of a `dispel4py` workflow using the `Multi` mapping with five processes.

Note that in this work, we also updated `dispel4py` from Python 2.7 to Python 3.10+, enhancing performance and leveraging the latest Python features.

[2]https://github.com/StreamingFlow/d4py
[3]https://docs.python.org/3/library/multiprocessing.html

```python
class IsPrime(IterativePE):
    def __init__(self):
        IterativePE.__init__(self)
    def _process(self, num):
        # this PE consumes one input and produces
            one output
        if all(num % i != 0 for i in range(2, num)):
            return num
```

Listing 1: `IsPrime` PE checks whether a given number is prime and returns the number if it is.

Listing 1 provides the code for the `IsPrime` PE in the 'isprime_wf.py' workflow (used in Figure 5a). The core functionality of this PE is within the `_process` function, where the prime-checking logic is implemented.

B. Serverless Computing

Serverless computing abstracts server management, allowing developers to focus on writing code. It automatically scales resources and charges based on execution time, making it cost-effective for applications with variable workloads. Key benefits include automatic scaling, reduced operational costs, and simplified deployment processes. However, it also introduces challenges such as cold start latency, limited execution duration, and complexities in state management and inter-service communication.

C. Language Models and Transformers

Advanced transformer-based natural language processing models have revolutionized code understanding and generation. They are used for tasks such as semantic code search, summarization, and completion. Notable examples include `CodeT5` [Wang et al.(2021)], `UniXcoder` [Guo et al.(2022)], and `ReACC-py-retriever` [Lu et al.(2022)]. These models, leveraging large-scale pre-training and fine-tuning, have been selected for `Laminar 2.0` following extensive evaluation in our previous work [Zahra et al.(2023)].

D. Semantic Code Search and Code Recommendation

Efficient semantic code search in `Laminar 2.0` enhances developer productivity by simplifying the discovery of relevant code snippets and workflows. In serverless environments, effective search capabilities are crucial for managing and reusing code components, significantly streamlining the development process. Code search can be categorized into *text-to-code* and *code-to-code* searches. *Text-to-code search* involves retrieving code that is semantically similar to a given text-based description. This approach uses advanced natural language processing (NLP) and machine learning models to understand the context and meaning of the text input and find relevant code snippets that match the described functionality. In `Laminar 2.0`, this capability is implemented using `CodeT5` and `UniXcoder` advanced deep learning models (see Section V).

Code-to-code search identifies similar code based on an input code snippet. This search is useful for code completion and clone detection and can be performed in three ways:

1) **Syntactic similarity:** Finds identical or nearly identical code snippets, similar to a text editor search.

2) **Semantic similarity:** Identifies functionally equivalent code segments with minor differences, useful for finding code clones.
3) **Structural similarity:** Compares the overall structure of code snippets rather than focusing on exact syntax. It is ideal for completing partial code snippets and making code recommendations.

`Laminar 2.0` employs **structural similarity** through the integration of `Aroma` (see Sections II-E and VI).

E. Aroma

`Aroma` [Luan et al.(2019)] [4] is a code recommendation tool for discovering relevant code snippets in large codebases. Originally for Java, `Aroma` uses structural similarity with simplified parse trees (SPTs) to compare snippets, identifying common coding patterns.

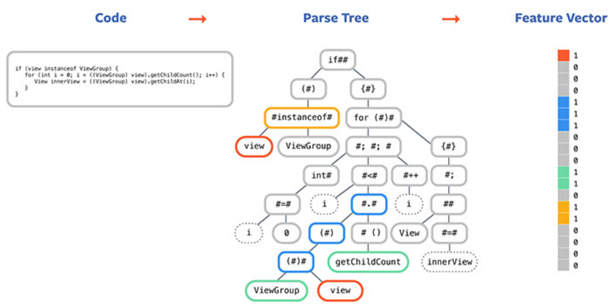


Fig. 2: Code to Parse Tree to Feature Vector

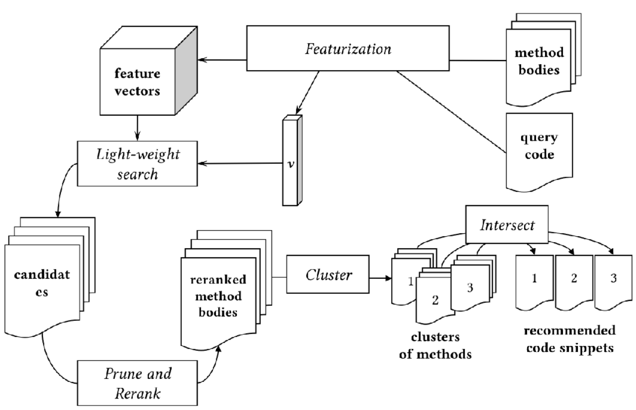


Fig. 3: Aroma code recommendation pipeline

As shown in Figure 2, `Aroma` converts code snippets into parse trees [Candillon(2008)]. The subsequent steps in Figure 3 ensure efficient code recommendations.:

- **SPT Generation:** Converts parse trees into streamlined representations using ANTLR (Section II-F), preserving structure while abstracting non-essential details.
- **Feature Extraction and Search:** Generalizes variable names and encodes context, using matrix multiplication for quick snippet identification.
- **Prune and Rerank:** Eliminates irrelevant segments and reranks results based on structural similarity.
- **Clustering:** Groups similar snippets with iterative clustering, enhancing recommendations.
- **Creating Recommendations:** Prunes a snippet against others in its cluster to form the final recommendation.

F. ANTLR: ANother Tool for Language Recognition

ANTLR [Parr(2013)] is a tool that generates parsers from grammar definitions to parse languages, building parse trees that provide a structural representation of the code. In `Laminar 2.0`, we used ANTLR to generate our own parsers for Python code. These parse trees are then transformed into SPTs. The process includes: a) *Grammar Definition* to define the syntax rules and structure of the target language; b) *Parser Generation* where ANTLR generates a parser and lexer based on the grammar definition; and c) *Parsing Code* where the parser processes input code to create a parse tree.

III. Laminar 2.0 Overview

`Laminar 2.0` enhances the original framework with new features for serverless `dispel4py` streaming workflows. It supports user operations like registering users, workflows, and PEs; running workflows in various modes; listing registry contents; updating descriptions; and performing advanced semantic or literal searches. Additionally, it offers code recommendations, making it a versatile tool for developers and researchers. `dispel4py` automatically parallelizes workflows, enabling `Laminar 2.0` to support seamless and efficient parallel execution.

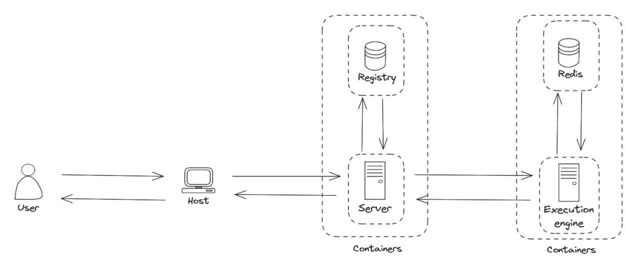


Fig. 4: Laminar 2.0's new architecture with containerisation

The key components of `Laminar 2.0`—the client[5], server[6], registry, and execution engine[7]—have undergone substantial enhancements. Container management has been integrated for easier deployment and scalability. The architecture, shown in Figure 4, fully supports Python 3.10+, allowing users to utilize the latest features of `dispel4py`.

The **client** provides a user-friendly interface for registering and managing PEs and workflows, performing semantic searches, running workflows, and retrieving context-aware code completions. Enhancements improve usability and functionality. The **registry** stores detailed metadata for users,

[4] https://github.com/facebookresearch/aroma-paper-artifacts/tree/main

[5] https://github.com/StreamingFlow/dispel4py-client/tree/main
[6] https://github.com/StreamingFlow/dispel4py-server
[7] https://github.com/StreamingFlow/dispel4py-execution

PEs, and workflows, facilitating advanced search capabilities and efficient workflow management. The **server** coordinates system functionality, organized into layers for controllers, services, models, and data access. It handles client requests, manages resources, and supports dynamic workflow execution, improving resource management. Finally, the **execution engine** executes workflows serverlessly, supports auto-import mechanisms for dependency management, and operates in local and remote environments with minimal configuration.

IV. Laminar 2.0 Core Enhancements

This section introduces the key enhancements of the main components that mark the evolution to `Laminar 2.0`.

A. Client: Client Functions

`Laminar 2.0` significantly enhances the client-side interface, streamlining usability by automating complex mapping parameters. Users can now execute dynamic workflows with simplified commands. For example, a dynamic workflow with five iterations and processes can be initiated with a single command, as demonstrated in Listing 3. This process was more difficult in `Laminar 1.0`, as shown in Listing 2. The execution engine now automatically optimizes the number of processes, which can be adjusted in the configuration settings.

```
client.run(graph, input=5, process=Process.DYNAMIC,
    \
    args=edict({'num':5, 'iter':5, 'simple':False, \
    'redis_ip':'localhost', 'redis_port':'6379'}))
```

Listing 2: Running a workflow dynamically in Laminar 1.0

```
client.run_dynamic(graph, input=5)
```

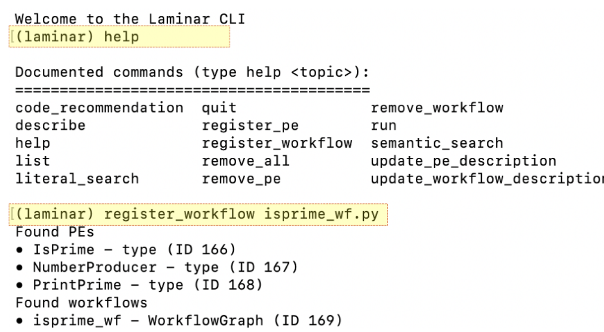
Listing 3: Running the same workflow in Laminar 2.0

These changes significantly reduce code complexity and enhance usability. `Laminar 2.0` also introduces parallel execution options, including static workload distribution with *multiprocessing* and dynamic distribution with *Redis*, ensuring efficient workflow execution across diverse environments. Table I lists all the available client functions in the framework. Examples using these functions are available at [8].

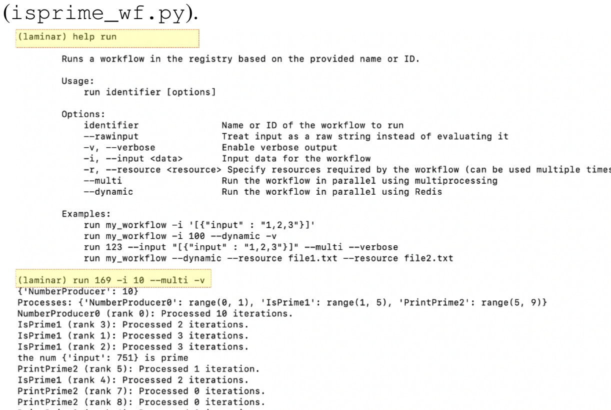
[8]https://github.com/StreamingFlow/dispel4py-client/tree/main/CLIENT_EXAMPLES

Function	Description
register	Registers a new user
login	Logs in an existing user
register_PE*	Registers a new PE
register_Workflow**	Registers a new workflow
get_PE	Retrieves a PE by name or ID
get_Workflow	Retrieves a workflow by name or ID
get_PEs_By_Workflow	Retrieves all PEs associated with a workflow
get_Registry	Retrieves all items in the registry
describe	Provides a description of a PE or workflow
update_PE_Description*	Updates a PE's description
update_Workflow_Description*	Updates a workflow's description
remove_PE	Removes an existing PE
remove_Workflow	Removes an existing workflow
remove_All*	Removes all PEs and workflows
search_Registry_Literal**	Performs a literal search
search_Registry_Semantic**	Performs a semantic search
code_Recommendation*	Performs a code recommendation
run**	Executes a workflow sequentially
run_multiprocess*	Executes a workflow in parallel
run_dynamic*	Executes a workflow using REDIS

TABLE I: Client func.: *new functions, **improved functions



(a) `CLI: help` command and registering a workflow (`isprime_wf.py`).



(b) `CLI: help run` command and running the workflow (ID 169) in parallel with *multiprocessing*.

Fig. 5: `CLI`:(a) Registering a workflow; (b) Running a workf.

B. Client: Command Line Interface (CLI)

To further simplify user interactions, a new `CLI` was introduced in `Laminar 2.0`, as shown in Figures 5a and 5b. The `CLI` allows users to search, register, and run workflows easily, providing functionalities for managing the registry and executing workflows. In Figure 5a, the `isprime_wf.py` generates a user-defined number of random numbers (e.g. `-i 10`, generates 10 numbers) and prints only the prime ones.

The `CLI` offers commands like `remove_all` to delete all registered PEs and workflows, and `help` to list commands, enhancing ease of use. Conversely, client functions (Table I) offer granular control, enabling script or Jupyter notebook integration for tasks like registering, removing, and describing PEs and workflows. Users can interact with `Laminar` via the `CLI` for command-line tasks or client functions of Table I for more complex scripting and automation. Instructions for both methods are provided in the User Manual available at [9].

C. Client: Automatic Descriptions

`Laminar 1.0` automatically generated PE descriptions using the `CodeT5` Language model when not provided by users, crucial for advanced search functionalities (see Sections V). `Laminar 2.0` improves this by utilizing the full PE class context instead of just the PE `_process()` method (where the logic of a PE is programmed), resulting in more accurate descriptions. It also extends automatic description generation to workflows, creating a class named after the workflow and including all PE functions as methods for comprehensive descriptions. Users can update these automatically generated descriptions via the `CLI` or client functions (see Table I), with changes reflected in the registry.

D. Registry: Database Improvement

To enhance the stability and scalability of `Laminar 2.0`, the database schema was updated to efficiently store larger datasets. The registry now uses *MySQL* to hold essential information about workflows and PEs. Previously, Python code was stored as a String field, which limited storage size. We have transitioned to character large objects for storing code and embeddings, accommodating increased data storage requirements and ensuring better performance and scalability.

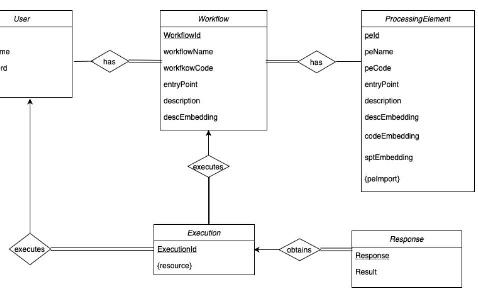


Fig. 6: Updated Database Schema

The database schema has been further normalized to eliminate redundancy and ensure data integrity. New attributes and tables have been introduced to enhance its structure. The updated schema, illustrated in Figure 6, shows the new tables and indexes to improve performance. Key elements of the new registry's database are summarized in Table II.

E. Execution Engine and Client: True-Streaming

A major improvement in `Laminar 2.0` is the shift from batch to stream-based communication between the client and execution engine, enhancing real-time data processing. In `Laminar 1.0`, the engine used HTTP/1.1, running the entire workflow, capturing the output to `stdout`, and sending the complete response back to the client, which was inefficient for real-time processing.

`Laminar 2.0` now leverages HTTP/2 streaming, allowing independent, bidirectional frames between client and server[10]. This ensures efficient, real-time data processing and minimizes latency by sending outputs as they become available. The execution engine uses *Flask*'s response streaming, transferring `stdout` to a concurrent queue, enabling real-time workflow output reading and line-by-line streaming to the client. The client was also adapted to receive this stream data.

Table Name	**Description**
User	Stores user information. Each user can be associated with multiple workflows, ensuring a one-to-many relationship.
Workflow	Contains details about each workflow. Each workflow can have multiple PEs and can be executed multiple times by different users.
Processing Element	Stores information about the processing elements. PEs are reusable components that can be associated with multiple workflows.
Execution	Tracks the execution of workflows. It includes execution-specific details. Each execution record is linked to a workflow and user.
Response	Captures the results of workflow executions. Each response is linked to a specific execution.

TABLE II: Key Elements of the Updated Database Schema

F. Server and Execution Engine: Resource Management

In `Laminar 1.0`, managing resources required by workflows in `dispel4py` posed several challenges. Resources, such as input files or other necessary data, were transferred to the execution engine by serializing a directory named `resources/` and including it in the HTTP request to the server. This approach necessitated manual management of the `resources/` directory for each workflow execution, leading to repeated transmission of potentially large files.

`Laminar 2.0` streamlines this process by allowing users to specify a list of required resources with the execution request. The server checks its cache for these resources and, if any are missing, responds with a `resources` message detailing the required files. New endpoints on the execution engine and server accept HTTP multipart requests for these files. Upon receiving the resources, the execution engine verifies their presence and proceeds with workflow execution. This method eliminates the need for a dedicated `resources` directory and allows direct file path specification, improving transparency and easing debugging. Additionally, a new caching mechanism reduces the need for retransmitting large files, optimizing the resource management process.

V. Advanced Search Features

`Laminar 1.0` provided advanced features like literal term search, semantic code search, and code completion. In `Laminar 2.0`, we have enhanced these features to improve user experience and efficiency. We evaluated several methods for code and text search, selecting the most effective approaches to ensure robust search functionalities.

A. Literal Searches

`Laminar` supports literal searches, enabling users to find workflows and processing elements (PEs) by matching search terms in their names or descriptions. This highlights the importance of having detailed descriptions for all PEs and

[9]https://github.com/StreamingFlow/dispel4py-client/wiki
[10]https://www.rfc-editor.org/info/rfc9113

workflows as introduced in Section IV-C. Figure 7 shows an example of a literal search for the term ‘words’ in both PEs and workflows, displaying the matching results from the registry.

```
(laminar) help literal_search
Searches the registry for workflows and processing elements matching the search term in the name or description.

Arguments:
  search_type   Type of items to search for. Choices are:
                - 'workflow': Search only for workflows
                - 'pe': Search only for processing elements (PEs)
                - 'both': Search for both workflows and PEs (default)
  search_term   The term to search for in the registry.

Usage:
  literal_search [workflow|pe|both] [string]

Examples:
  literal_search workflow some_term
  literal_search pe some_term
  literal_search both some_term
(laminar) literal_search both "words"

REGISTRY

Result 1: ID: 181
PE Name: CountWords
Description: CountWords - Count the number of words in the sequence.

Result 2: ID: 183
PE Name: SplitWords
Description: SplitWords class.

Result 3: ID: 184
Workflow Name: wordcount_wf
Description: A workflow that counts the number of words received as an input list
```

Fig. 7: Search for the term ‘words‘ in both PEs and workflows.

B. Semantic Code Searches

Semantic code search, or *text-to-code search* (see Section II-D), involves retrieving code semantically similar to a text-based description. In `Laminar 2.0`, this capability leverages two advanced deep learning models: `CodeT5` for generating descriptions and `UniXcoder` for embedding them. When a user registers a workflow (along with its PEs) or a PE directly, `Laminar` automatically generates their descriptions if not provided and creates normalized description embeddings using `UniXcoder`, which are then stored in the registry. The choice of this model was validated in our previous work [Zahra et al.(2023)]. When a user performs a semantic code search, the system encodes the input query and compares it to precomputed embeddings of PEs and workflows stored in the registry. The core mechanism utilizes cosine similarity to measure the semantic closeness between the user’s query embedding and the descriptions’ embeddings of PEs or workflows.

```
(laminar) help semantic_search
Searches the registry for workflows and processing elements matching semantically the search term.

Arguments:
  search_type   Type of items to search for. Choices are:
                - 'workflow': Search only for workflows
                - 'pe': Search only for processing elements (PEs)
  search_term   The term to search for in the registry.

Usage:
  semantic_search [workflow|pe] [search_term]

Examples:
  semantic_search workflow some_term
  semantic_search pe some_term
(laminar) semantic_search pe "a pe that is able to detect anomalies"
Performing similarity search on pe, with query type: text
Encoding query as text...
   peId              peName                                    description  cosine_similarity
5   177  AnomalyDetectionPE                             Anomaly detection PE.           0.740170
4   176          AlertingPE                               AlertingPE class.           0.448650
0   171             IsPrime  A pe to check if the input number received is ...           0.302320
6   178     NormalizeDataPE  This pe normalizes the temperature of a record...           0.260940
3   175     AggregateDataPE  AggregateDataPE - Aggregate data from a sequen...           0.257947
```

Fig. 8: Semantic search for PEs using a descriptive query.

The Figure 8 shows an example of a semantic search, where the term ‘a pe that is able to detect anomalies’ is used to find relevant PEs. The semantic search process, begins by normalizing the response data and encoding the user’s query. The similarity scores are computed, and the results are sorted to identify the top matches. By default, the system returns the top five results, but this can be configured as needed.

VI. Aroma for Laminar

In `Laminar 1.0`, *code-to-code* search was implemented using the `ReaCC-py retriever` model, which excelled at clone detection by recalling functions from identical or semantically equivalent code. However, this approach was limited in aiding the development process, as it primarily focused on identifying identical existing PEs based on provided code. To enhance code recommendation capabilities, we integrated `Aroma` (introduced in Section II-E), a tool designed to provide developers with recommendations of existing functions based on partial code snippets. This approach better assists developers by allowing them to see completed PEs that contain code similar to their snippets. The integration required adapting `Aroma` to parse Python code into simplified parse trees (SPTs) using ANTLR (see Section II-F). Python ANTLR lexers and parsers are now available in our source code [11].

When a PE is registered, the client automatically extracts the full class definition, and the source code is then parsed into an SPT, which is a parse tree that abstracts away non-essential details while preserving the hierarchical structure. Features capturing the syntactic and structural elements of the code are extracted from the SPT and stored in the registry as embeddings in JSON format (see ‘sptEmbedding’ in Figure 6). These embeddings enable `Laminar` to compare code snippets and provide recommendations based on structural similarity.

A. Code Recommendation Mechanism

When a code recommendation is initiated, the input query is parsed into an SPT, and features are extracted. These features are compared against stored PE ‘sptEmbedding’ using cosine similarity to identify the top similar PEs. The results are ranked based on similarity scores and formatted to display details such as name, description, and code snippets. Unlike the original `Aroma` algorithm, our implementation uses cosine similarity for efficiency, simplicity, and scalability, without the need for complex clustering or reranking steps. By default, `laminar` returns up to five PEs with a similarity score above 6.0, a configurable parameter.

```
(laminar) help code_recommendation
Provides code recommendations from registered workflows and processing elements matching the code snippet.

Arguments:
  search_type   Type of items to search for. Choices are:
                - 'workflow': Search only for workflows
                - 'pe': Search only for processing elements (PEs)
  code_snippet   The code_snippet to get recommendations from the registry.

Options:
  --embedding_type  The type of embedding to use. Choices are:
                - 'spt': Perform a search based on SPT features
                - 'llm': Perform a search based on LLM-generated embeddings

Note: code recommendations for workflows only possible with 'spt' embedding_type

Usage:
  semantic_search [workflow|pe] [code_snippet] [--embedding_type llm|spt]

Examples:
  code_recommendation pe code_snippet --embedding_type spt
  code_recommendation workfkow code_snippet --embedding_type spt
  code_recommendation pe code_snippett --embedding_type llm
(laminar) code_recommendation pe "random.randint(1, 1000)"

   peId          peName                   description  score                        simlarFunc
0   172  NumberProducer  The number producer class.    8.0  def _process(self, inputs):
[<module_name_1723108902_1.NumberProducer object at 0x31a987a90>]
(laminar) code_recommendation workflow "random.randint(1, 1000)"

   workflowId workflowName                        description  occurrences
0         174   isprime_wf  check if the num is prime or not            1
[<dispel4py.workflow_graph.WorkflowGraph object at 0x31a9878b0>]
```

Fig. 9: Example of using the `CLI` for code recommendation.

[11] https://github.com/StreamingFlow/dispel4py-client/tree/main/Aroma

For workflow code recommendations, the input query is parsed into an SPT, features are extracted, and similar PEs are identified. Workflows containing these PEs are retrieved, ranked by similarity, and detailed information is displayed. As shown in Figure 9, code recommendations can still use the `ReaCC-py retriever` model by specifying `--embedding_type llm`, but the default is `Aroma` ('spt'). Users can retrieve the source code of PEs or workflows using the `describe` command and their IDs.

VII. Evaluations

A. Dataset CodeSearchNet PE Creation

To facilitate a comprehensive evaluation of our system, we utilized the *CodeSearchNet* [Husain et al.(2019)] dataset, which contains a large collection of Python functions (450k) paired with their corresponding text descriptions. These functions were converted into PEs using ANTLR, ensuring compatibility with `Laminar`'s proprietary PE format. Additionally, PEs that were semantically similar, based on their textual descriptions, were grouped together. Each PEs in the new dataset was given a unique identifier to avoid ambiguity, particularly in cases where functions might have duplicate names. This new *CodeSearchNet PE dataset* served as the foundation for evaluating various functionalities within `Laminar`.

B. Description Generation

We evaluated the `CodeT5` model for generating PE descriptions in `Laminar`. In `Laminar 1.0`, descriptions were derived only from the `_process()` method, often resulting in insufficient context and poor performance. `Laminar 2.0` addresses this by expanding description generation to include the entire class definition, significantly improving relevance and quality, as shown in Figures 10a and 10b.

```
0                                            Process the sequence of unknown - sequence products.
1   Process the sequence of sequence of sequence of sequence of sequence of sequence of sequence of
2                                                          Generate a random n - token session.
3                                                       Generate a random seqno from 1 to 1000.
4                                                                               Log the nagios.
5   Process the sequence of sequence of sequence of sequence of sequence of sequence of sequence of
6                                           Process the nltk tag and return the average score of the tag.
7                       This method processes the nltk sentiment file and returns the average sentiment score and
8                         Process the nltk tag and return the article the average sentiment score and the method
9                           Process the nagios. txt file and return a dictionary with the total articles found.
```

(a) `CodeT5` descriptions from just the `_process` method.

```
0                                       A PE that produces a sequence of numbers from 1 to n.
1                                  This PE is a base PE for test 2 - in - one - out tests. It
2                                          Generate a random word from the sequence of words.
3                                   Producer PE for producing a random number from 1 to 1000
4                                                   A logger for the missing sequence sequence.
5                                          A PE that implements a multi - producer interface.
6                                                 AFASTA AFINNSentimeScore - AFASTA AFINNS
7     This PE calculates the average sentiment score based on the word sentiment file provided.
8                                              AFASTIME - SENTIME - SENTIME - SENTIME - S
9                        This PE reads the data from the twitter and writes it to the output file.
```

(b) `CodeT5` descriptions using the full PE class.

Fig. 10: Descriptions generated from different code contexts.

C. Semantic Code Searches Evaluation

We evaluated `Laminar`'s *text-to-code* search functionality introduced in Section V-B using our *CodeSearchNet PE dataset*. For each PE, descriptions were generated using `CodeT5`, and embeddings were created with `UniXcoder`, then stored in the system's registry for semantic search. The evaluation process involved automated querying with natural language descriptions from the original *CodeSearchNet* dataset. These queries were matched against the stored embeddings to measure retrieval effectiveness. Performance was assessed using precision and recall metrics: precision reflects the proportion of relevant PEs retrieved, and recall indicates how many relevant PEs were successfully identified. The F1-score provides a balanced measure of precision and recall.

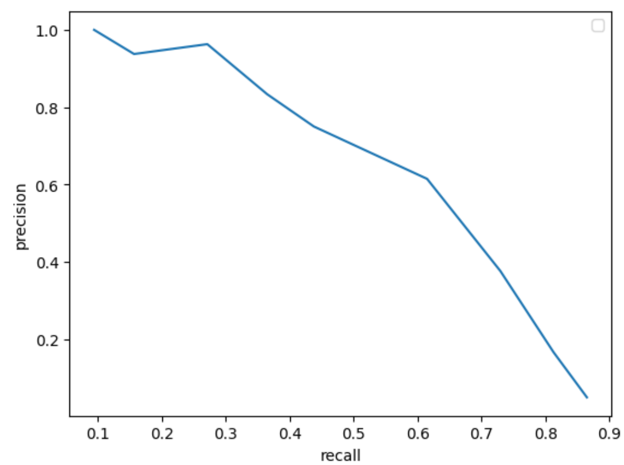


Fig. 11: Precision-recall for `text-to-code search`.

Figure 11 presents the precision-recall curve. Our method achieved a best F1 score of 0.61, indicating a balance in *text-to-code* search performance.

D. Code Recommendation Evaluation

To evaluate the *code-to-code* search capabilities in Section VI, we compared the `Aroma` algorithm with the previous `ReaCC-py` from `Laminar 1.0` using the *CodeSearchNet PE dataset*. We conducted experiments to assess the precision and recall of both models, using each PE as a query to test their retrieval effectiveness. To simulate real-world scenarios, we progressively reduced the input snippet sizes.

Figures 12 and 13 show that `Aroma` maintains high precision with full code snippets (0% dropped) and performs better with partial snippets (50% and 75% dropped), while `ReaCC-py retriever` exhibits a steeper precision decline as more results are retrieved and code is omitted. At 90% code omission, both models struggle, but `Aroma` still outperforms `ReaCC-py retriever`. Overall, `Aroma` achieved a maximum F1-score of 0.63, significantly higher than `ReaCC-py retriever`'s best of 0.24.

VIII. Related Work

Comparison with other serverless frameworks like FuncX [Li et al.(2022)], PyWren [Jonas et al.(2017)], Apache OpenWhisk[12], and Apache Flink [Katsifodimos and etc(2015)] highlights `Laminar 2.0`'s unique strengths

[12] https://openwhisk.apache.org/documentation.html

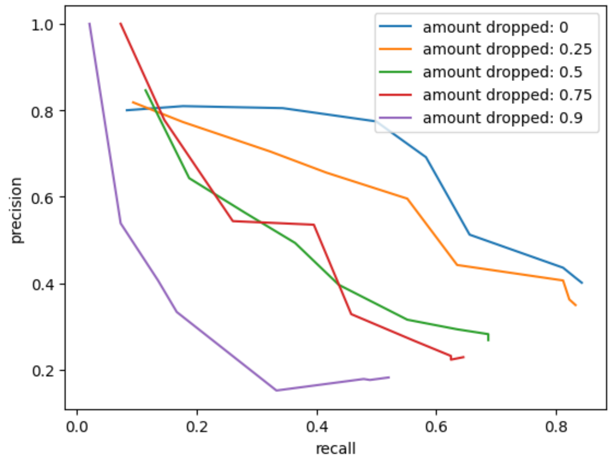


Fig. 12: Precision-recall for `Aroma`.

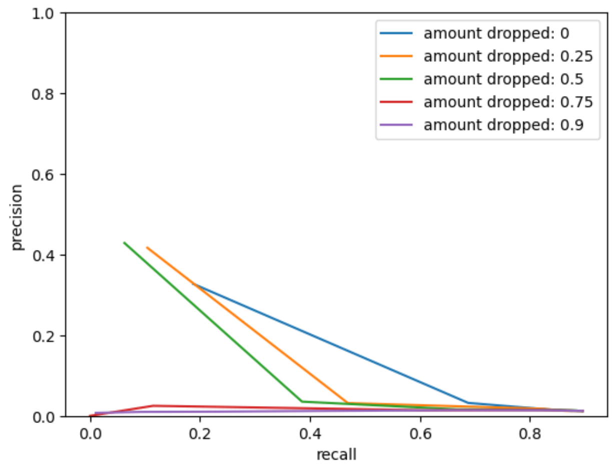


Fig. 13: Precision-recall for `ReaCC-py retriever.`

in handling streaming data and integrating deep learning models for advanced code search and completion. `Laminar 2.0` offers a more developer-friendly environment with its enhanced search and completion features, making it stand out in the domain of serverless computing for stream-based workflows. Furthermore, Senatus [Silavong et al.(2021)], an improvement to `Aroma`, further enhances structural code recommendation using Locality Sensitivity Hashing (LSH).

IX. Conclusions and Future Work

`Laminar 2.0` brings enhancements to the original framework, elevating its capabilities for managing serverless streaming workflows, advanced code searches and recommendations. With improved client functionality, full Python 3.10+ support, and a scalable, Dockerized architecture, `Laminar 2.0` offers a robust environment for developers. Future work will focus on supporting multiple execution engines, and refining deep learning models, including LSH for structural code.

References

[Candillon(2008)] William Candillon. 2008. Parse Tree. http://pecl.php.net/package/Parse_Tree.

[Eddelbuettel(2022)] Dirk Eddelbuettel. 2022. A Brief Introduction to Redis. arXiv:2203.06559 [stat.CO]

[Filgueira et al.(2014)] Rosa Filgueira, Iraklis Klampanos, and etc. Krause. 2014. dispel4py: A Python Framework for Data-Intensive Scientific Computing. *2014 International Workshop on Data Intensive Scalable Computing Systems* (2014). https://doi.org/10.1109/DISCS.2014.12

[Forum(1994)] Message P Forum. 1994. *MPI: A Message-Passing Interface Standard*. Technical Report. USA.

[Guo et al.(2022)] Daya Guo, Shuai Lu, Nan Duan, and etc. 2022. UniXcoder: Unified Cross-Modal Pre-training for Code Representation. In *Proceedings of the 60th Annual Meeting of the Association for Computational*. Association for Computational Linguistics, 7212–7225. https://doi.org/10.18653/v1/2022.acl-long.499

[Husain et al.(2019)] Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, and etc. 2019. CodeSearchNet Challenge: Evaluating the State of Semantic Code Search. *CoRR* abs/1909.09436 (2019). arXiv:1909.09436 http://arxiv.org/abs/1909.09436

[Jonas et al.(2017)] Eric Jonas, Shivaram Venkataraman, Ion Stoica, and Benjamin Recht. 2017. Occupy the Cloud: Distributed Computing for the 99%. *CoRR* abs/1702.04024 (2017). arXiv:1702.04024 http://arxiv.org/abs/1702.04024

[Katsifodimos and etc(2015)] P Carbone Asterios Katsifodimos and etc. 2015. Apache FlinkTM: Stream and batch processing in a single engine. *Bull. IEEE Comput. Soc. Tech. Comm. Data Eng* 36, 4 (2015).

[Kumar(2019)] Manoj Kumar. 2019. Serverless architectures review, future trend and the solutions to open problems. *American Journal of Software Engineering* 6, 1 (2019), 1–10.

[Li et al.(2022)] Zhuozhao Li, Ryan Chard, Yadu Babuji, and etc. 2022. Federated Function as a Service for Science. *IEEE Transactions on Parallel and Distributed Systems* 33, 12 (dec 2022), 4948–4963. https://doi.org/10.1109/tpds.2022.3208767

[Liang et al.(2022)] Liang Liang, Rosa Filgueira, Yan Yan, and Thomas Heinis. 2022. Scalable adaptive optimizations for stream-based workflows in multi-HPC-clusters and cloud infrastructures. *Future Generation Computer Systems* 128 (2022), 102–116. https://doi.org/10.1016/j.future.2021.09.036

[Liang et al.(2023)] Liang Liang, Heting Zhang, Guang Yang, and etc. 2023. Optimization towards Efficiency and Stateful of dispel4py. In *Proceedings of the SC '23 Workshops* (Denver, CO, USA) *(SC-W '23)*. 2021–2032. https://doi.org/10.1145/3624062.3624281

[Lu et al.(2022)] Shuai Lu, Nan Duan, Hojae Han, Daya Guo, and etc. 2022. ReACC: A Retrieval-Augmented Code Completion Framework. arXiv:2203.07722 [cs.SE]

[Luan et al.(2019)] Sifei Luan, Di Yang, Celeste Barnaby, and etc. 2019. Aroma: code recommendation via structural code search. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 152 (oct 2019), 28 pages. https://doi.org/10.1145/3360578

[Parr(2013)] Terence Parr. 2013. *The Definitive ANTLR 4 Reference* (2 ed.). Pragmatic Bookshelf, Raleigh, NC. https://www.safaribooksonline.com/library/view/the-definitive-antlr/9781941222621/

[Shafiei et al.(2022)] Hossein Shafiei, Ahmad Khonsari, and Payam Mousavi. 2022. Serverless computing: a survey of opportunities, challenges, and applications. *Comput. Surveys* 54, 11s (2022), 1–32.

[Silavong et al.(2021)] Fran Silavong, Sean J. Moran, and Antonios Georgiadis etc. 2021. DeSkew-LSH based Code-to-Code Recommendation Engine. *CoRR* abs/2111.04473 (2021). arXiv:2111.04473 https://arxiv.org/abs/2111.04473

[Wang et al.(2021)] Yue Wang, Weishi Wang, Shafiq Joty, and etc. 2021. CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation. https://doi.org/10.48550/ARXIV.2109.00859

[Zahra et al.(2023)] Zaynab Zahra, Zihao Li, and Rosa Filgueira. 2023. Laminar: A New Serverless Stream-based Framework with Semantic Code Search and Code Completion *(SC-W '23)*. 2009–2020. https://doi.org/10.1145/3624062.3624280