Team Stack Underflow

Tan Wei Liang
Pan Wen Zhuo, Ernest

Proposed level of achievement

Artemis

Link to download app (Android)

https://puu.sh/GaQgZ/101fbf7d13.apk

Motivation

Many modern programming languages offer a Read-Eval-Print-Loop (REPL) environment that allows users to interact directly with programs as they are being executed. The Python interpreter is one such example – code is executed line-by-line as the user enters them. This provides an excellent experiential learning opportunity to those who wish to learn such languages, as they can experiment and tinker with code on the fly.

However, these REPL systems share a common limitation – they often require access to a desktop operating system, be it a workstation or laptop. When the availability of the REPL environment is limited, this poses a major challenge to experiential learning, as experiential learning encourages active, continuous experimentation and observation.

In addition, one might want to simply explore and try new languages but does not wish to commit the effort into setting up the various SDKs and dependencies. For such cases, the setup requirements could be abstracted away from the user, allowing them to jump straight into coding without worrying about the dependencies.

Aim

We wish to promote experiential learning for programming by emulating REPL environments on mobile apps with minimal setup to make them more accessible.

In particular, we intend to target the following groups in NUS:

- CS1010S/E students (learning Python)
- CS1010J/CS2030 students (learning Java)
- CS1101S students (learning Source/JavaScript)
- CS1010/CS2100 students (learning C)

User Stories

1. As a beginner programmer, if someone sends me a code fragment and I do not know what it does, then I want to be able to evaluate it quickly to learn about it.

2. As a novice programmer, I want to be able to test my understanding of the language on the go by writing tricky code snippets and executing them to verify that my knowledge and intuition is correct.

3. As a veteran programmer, I want to be able to share interesting code snippets and discuss potentially unintuitive code execution results with others, in order to improve the overall programming experience.

Project Components

Types of Execution

Our project should be able to handle a few ways of evaluating programs. For brevity we will assign arbitrary letters to them and refer to those letters throughout the rest of the proposal.

Core Feature: REPL (Type A Execution) ✅

The project must be able to simulate a REPL environment that users can interact with. User input is sent to the runtime, and output is immediately returned to the user.

Core Feature: Pre-compiled Interactive (Type B Execution) ✅

More complex programs may be given in the form of source code files instead of simple text. In such cases, a REPL environment is no longer the most convenient solution. It would be more practical to simply compile the source code (for compiled languages) and run the program directly. Input from the user will be passed as standard input to the runtime while standard output will be passed back to the user.

Extension: Batch Processing (Type C Execution) ❌

If files are provided under Type B executions, users may optionally specify an additional text file representing standard input to the program. The program will then run completely autonomously, and the user can check back on the output of the program later.

Frontend (Mobile App)

Users will interact with the system through a mobile app. The app will be developed using Flutter rather than native APIs to avoid maintaining separate codebases for each mobile platform.

Core Feature: Runtime selection ✅

We aim to support multiple programming languages to cater to different types of users. They will be able to specify through the app exactly which language they would like to emulate, as well as which type of execution to use.

Core Feature: User input/output ✅

While the runtime is active, the app is responsible for capturing user input to be sent and program output to be displayed. Users will be able to provide line by line input similar to an actual interactive program execution, and have options available to reset or terminate the program.

Core Feature: File storage ✅

Users will be able to save their program code and input/output (if any) as text files in a specified location for future references. They will also be able to load saved code from any file location on their device, for Type B and Type C executions. We intend for the app to load the source code in plaintext to allow for a range of sources, either saved through the app or downloaded from an external location.

Extension: Autosave ✅

If users exit the app in the middle of writing a program, the unfinished program's code will be automatically saved and loaded for the user to continue when they next launch the app.

Extension: Sharing code and results ✅

Users may come across interesting or counterintuitive results when running programs and want to share the execution results. They may be a beginner programmer seeking clarification, or a veteran programmer looking to discuss obscure nuances of the language. We would like to provide a convenient way for them to communicate such results to others.

Extension: Keyword inventory ❌

For ease of writing code using a mobile device, we plan to implement shortcuts for commonly used keywords, such as language-specific reserved words and user-defined variable names. Users will be able to select keywords from an inventory and drag them into place anywhere in the code.

Backend (API)

Users' devices can vary wildly in hardware and software capabilities. There is no guarantee that the user's device is able to execute the given program in a desirable manner. As such, computation will be handled on the cloud, using well-known serverless options such as AWS to scale processing power to meet demand.

Core Feature: API Endpoint ✅

The backend API will feature endpoints to facilitate communication with the frontend app. As AWS is not yet well-supported on the Flutter platform itself, we will be using a WebSocket API so that the frontend app only needs to know how to make the appropriate requests.

We plan to use Amazon API Gateway coupled with AWS Lambda to handle incoming requests. The API will eventually be secured with an API key to ensure that only authorized apps are allowed to make requests.

Core Feature: Sandboxed Execution ✅

Arbitrary code execution is a major security concern. To address this, user-submitted code will be sandboxed in Docker container instances to prevent malicious input from affecting the platform running the program. The containers will be managed using AWS Elastic Container Service (ECS), triggered from the API endpoints.

Extension: Timeouts ✅

If users exit the frontend app gracefully, the app should send a request for the backend to terminate any running containers to avoid unnecessary consumption of cloud resources. However, if the app exits unexpectedly (e.g. power loss, force close, out of memory) such a signal may not be present.

Hence, if any container is inactive for an extended amount of time, it should be automatically killed to keep operating costs down. This applies also to users who leave a REPL terminal open and inactive for too long, as it drains valuable cloud resources. In such a case, the user will have to restart the terminal session to continue.

Supported Languages/Runtimes

Language	Type A	Type B	Type C
Python	Python interpreter	Python interpreter	Python interpreter
Java	JShell	Java compiler	Java compiler
C	igcc	gcc	gcc
JavaScript	Node.js	Node.js	Node.js
Source	js-slang	js-slang	js-slang

Core Feature: Python ✅

Python is taught in CS1010S and CS1010E to non-Computing students who may not be well-versed in programming in general. It is also a highly popular language around the world, which provides us with a massive potential user base.

Since Python is an interpreted language, Type A executions can be easily carried out using the Python interpreter itself. The same interpreter also supports Type B and Type C executions.

Core Feature: Java ✅

Java is taught in CS1010J and CS2030 which is again a large number of students and potential users.

Although Java is usually considered a compiled language, since JDK 9 an interactive REPL tool (JShell) has been bundled with the Java compiler, allowing us to perform Type A executions. Type B and Type C executions can be done using the standard Java compiler to compile and run the program.

Core Feature: C ✅

C is taught in CS1010 and CS2100.

C is a rather low-level compiled language, and there is no official REPL tool for it. However, we can make use of third-party programs such as igcc to help us run Type A executions. For Type B and Type C executions, a simple C compiler such as gcc should be sufficient for our needs.

Core Feature: JavaScript ✅

JavaScript is widely used by web developers all around the world.

Although most JavaScript is run through a web browser, we can use Node.js, a popular standalone JavaScript runtime environment, to perform all three execution types.

Extension: Source

Source is a sublanguage of JavaScript, used in the teaching of CS1101S. However, as of now Source is not used anywhere else, making support for Source a niche feature limited only to CS1101S students.

Thus, we have decided to lower the priority of Source from core feature to extension, and instead implement support for JavaScript as a core feature.

Source is implemented and maintained through the js-slang project. Its runtime is similar to that provided by Node.js, allowing us to perform all three execution types.

Extension: Others

There are many other languages (e.g. Ruby, Go, Haskell) that have potential to be implemented in the project. However, in the interest of time we will focus first on the languages described above.

In theory, once the framework for the project has been set up it should be easy to extend to new language options and implementations, so such work can in fact be achieved after the conclusion of the project as well.

Competitor Analysis

We have found existing solutions that are similar to what we want to achieve, but do not offer all of the features described above.

We plan to take reference from the strengths of the examples highlighted below, to create an app that can provide all the required features together.

Pydroid 3 - IDE for Python 3

(https://play.google.com/store/apps/details?id=ru.iiec.pydroid3)

There are numerous apps that claim to be IDEs, offering a rich set of development tools. For interpreted languages like Python, a REPL runtime environment may be provided as well.

However, these apps are highly specific to a single language. If the user wants to try out another language, they will need to install many different apps, which can be cumbersome.

Compiler

(https://play.google.com/store/apps/details?id=app.compiler)

This mobile app offers support for a wide range of languages. It also has a well-designed user interface, with tools to select local source code files or to type common programming symbols.

However, it only supports batch processing. The entire program must be provided before runtime, and the interactive REPL environment is not available.

Ideone

(https://ideone.com/)

Unlike the previous two examples, Ideone is a web-based application. Its user interface is simple and concise, allowing users to jump straight into what they want.

Unfortunately, it also only supports batch processing. The source code can be edited between executions, but not during runtime.

Tech Stack

1. Flutter
2. Assorted compilers and interpreters for the supported languages
3. Docker
4. Various Amazon Web Services (AWS) products, illustrated in the diagram below

Diagrams

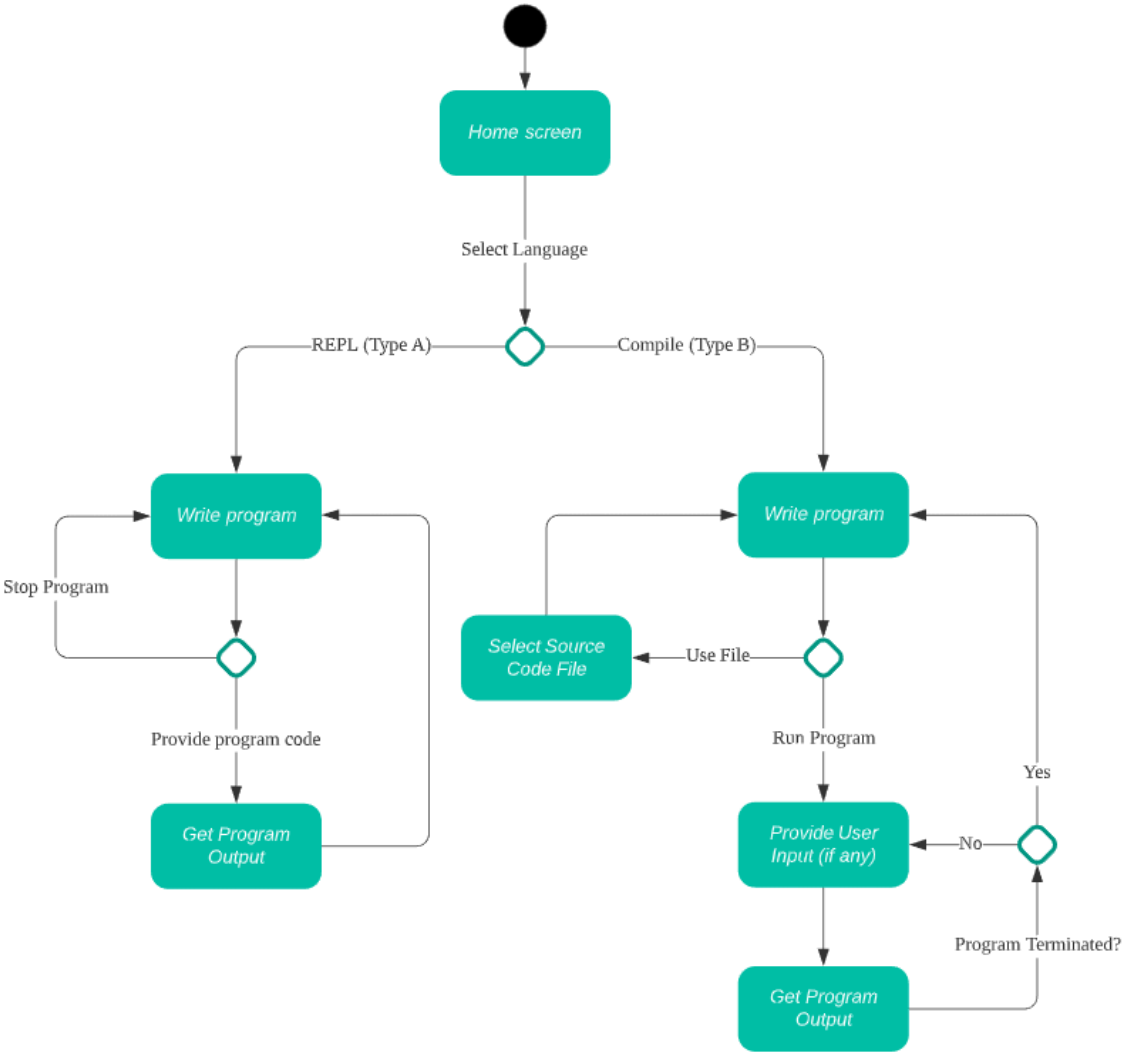

Figure 1: Frontend activity diagram

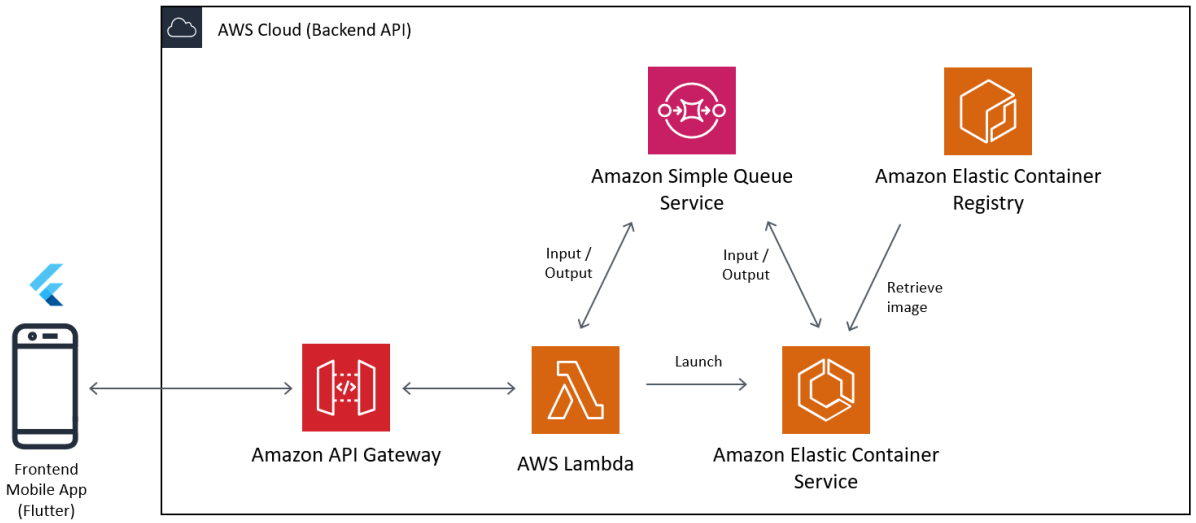

Figure 2: Backend architecture diagram

Timeline

Week	Date	Details
0	11/5 – 17/5	Liftoff · Q&A, get to know more details about the programme · Apply for mentorship · Ignition
1	18/5 - 24/5	· Investigate feasibility of backend architecture · Discuss plans for frontend UI design · Construct proof-of-concept
2	25/5 - 31/5	· Set up frontend project · Set up backend project · Prepare Milestone 1 deliverables
3	1/6 – 7/6	Milestone 1 · Evaluate other teams · Evaluate own progress
4	8/6 – 14/6	· Develop prototype frontend · Develop prototype backend
5	15/6 – 21/6	· Continue development on frontend and backend · Set up API between frontend and backend
6	22/6 – 28/6	· Continue development on frontend and backend · Perform functional testing · Draft a user guide · Prepare Milestone 2 deliverables
7	29/6 – 5/7	Milestone 2 · Evaluate other teams · Evaluate own progress
8	6/7 – 12/7	· Continue functional testing and bug fixing · Add unit tests · Decide on which extensions to complete · Set up Continuous Integration to run the unit tests
9	13/7 – 19/7	· Write documentation for users and potential future contributors · Perform user testing based on a small group (friends) · Begin implementing extensions
10	20/7 – 26/7	· Implement changes based on user feedback · Continue development on extensions

11	27/7 – 2/8	Milestone 3 · Evaluate other teams · Evaluate own progress
12	3/8 – 9/8	· Finalize marketable product · Set up Continuous Deployment
13	10/8 – 16/8	Next semester begins · Advertise to target audience (students) during lectures
14	17/8 – 23/8	· Prepare for Splashdown
15	26/8	Splashdown

Progress (Milestone 2)

Our work-in-progress components can be found across various repositories on GitHub under https://github.com/team-stack-underflow.

Frontend

We have completed a basic prototype app in Flutter, which is demonstrated in the video.

A user guide can be found here. The app's code base can be found here.

As of Milestone 2, basic features essential to the identity of the app have been developed. It is able to capture user input and effectively communicate commands to the backend.

Backend

The backend is now a WebSocket API, where it was previously a REST API. This is to provide ad-hoc asynchronous communication between user interface and program.

A detailed description of the API's features can be found here. The API allows any WebSocket client to access Runnable's features — if a developer so chooses they may develop their own frontend (e.g. Telegram chat program) with an appropriate API key.

The container images used for the sandboxed execution can be found here.

Problems Encountered

We faced problems that were inherent limitations of the technologies we used (Flutter and AWS).

Websocket connection issues

As Flutter is a relatively new framework, there was not much documentation on the libraries and packages concerning web sockets. As such, a significant amount of time was spent troubleshooting issues with the web socket functionality. We also realized that we had to maintain one single websocket connection for the lifetime of the UI widget, and had to adjust our backend to cater for it.

Container interaction issues

For AWS, we found that there were a number of restrictions on what we could do with a running container instance as it is in its own private network and is normally unable to connect to the external Internet. This made us redesign our backend architecture.

App unable to perform dynamic animation

We had difficulty creating a widget that would change size depending on if it was selected or not, without rebuilding the entire page. This created an issue with the streambuilder widget on our Compile page as it caused the code output to be duplicated when it was rebuilt. This was solved using listeners to only rebuild the widget when necessary.

Progress (Milestone 3)

Enhancements / Extensions

Frontend UI enhancements

We have designed a suitable app icon that we believe fits the spirit of the app as well as worked it in as part of the interface in certain places.

Our frontend app has been extended with some additional convenience features to enhance the user experience, such as the ability to save and load files and to share code with others.

Several settings have been added that enable the user to change based on their preferences, such as theme and font size.

Backend optimizations

Our backend has also been improved to be more robust, and responds to clients in a more deterministic manner. To address the problem raised in the previous milestone, it was further reworked to better handle multiple running instances in a single connection.

To prevent excessive usage of backend resources, the backend has been enhanced with a simple API key authentication. The API key is only required to launch new container instances, as all other resources are dependent on the launch command.

In our initial implementation for Milestone 2, we used a standard message queue to store communication between the client and program. However, due to the decentralized nature of

Amazon SQS there were a few occasional bugs where messages were either missing or appearing out of order.

This was solved using first in first out (FIFO) message queues. A FIFO queue preserves the order of messages and also enforces 'exactly-once' processing, so duplicate inputs and outputs are not mistakenly ignored.

Project Management

We have used GitHub as our version control and issue tracker. At each Orbital milestone, a pre-release is made to capture the snapshot of the project at that point in time.

Upcoming features and bug fixes are documented in the form of issues and pull requests. Any new extension or bug found is first described in a new issue. Thereafter, a new branch is created with the updated code and a pull request to master is opened. It is then merged after a code review and some functional testing.

CI/CD tools are employed at various areas throughout the project. For the backend, GitHub Actions has been set up to automate the building and deployment of container images to AWS whenever changes are made to the dockerfiles, as it can be time-consuming to do locally every time. For the frontend, GitHub Actions has been set up to run automated widget tests as well as automatically build and upload an Android APK file that can be installed to do manual functional testing.

We have also created several pages on the runnable-frontend GitHub wiki (here) to document the organisation and implementation of our code. A similar description for the backend can be found here (note that the documentation spans multiple wiki pages across different repositories).

Testing

Functional Testing

For functional testing, we have installed the frontend app on various Android emulators (such as the Android Emulator through Android Studio and Bluestacks 4) as well as on our own personal phones. This is to ensure that the app meets all of the core features as described above.

Automated Testing / Regression Testing

Flutter provides three main types of automated tests. We found that widget testing is the most appropriate type of testing to perform given the scope of our project. Our code does not have enough modular parts to make unit testing effective, and integration testing would require too much work to set up and maintain.

These tests can be run locally from the command line, and will also run automatically when a commit is pushed or a pull request is opened on GitHub. This will help us to monitor any unexpected regressions when writing features and fixes.

User Testing

Our third method of testing involves rolling out a beta version of the app to a small handful of potential users to gain first hand feedback on what can be improved.

Testers were given a list of questions to answer about their experience with the app. The form can be found here. In some cases, we interviewed the testers directly and obtained verbal feedback instead.

Based on the feedback given, we made minor adjustments to some UI elements to make the user experience more intuitive. We have also noted down some popular / highly requested features, and while we may not be able to implement them in time for Milestone 3 due to the technical complexity involved, we will do our best to look into them until Splashdown (and possibly beyond that as well).

Problems Encountered

App unable to read/write to phone storage

When trying to package our app for release we encountered some platform-specific issues with regards to permissions management, which was not present in our development environment emulator. While most of it has been resolved by using the permission_handler flutter package, there are still some edge cases (such as not being able to write to external storage) that we are still unable to troubleshoot, mainly due to a lack of documentation on the Flutter platform.

Code takes too long to process

Our backend also has some issues regarding performance. Before Milestone 2 we prioritised a minimum viable product over performance optimizations, and hoped to improve on it in Milestone 3. Our main issue is a long waiting time to initialize a container instance. However, so far our attempts at optimization have proved unsuccessful. We have tried increasing processor limits on AWS, as well as using a smaller Docker image based on Alpine Linux instead of Ubuntu, but neither solution saw any significant performance improvement.