

Visual Programming Language for Web

(Student Proposed)

Oisín Canty

Final Year Project
BSc in Computer Science

Supervisor: Dr Jason Quinlan
Second Reader: Dr Frank Boehme

Department of Computer Science
University College Cork

25th April 2022

Abstract

In this project we will explore the design and implementation of a visual programming language platform. Visual programming languages have existed for decades but often suffer from an identity crisis. In recent years, visual workflow automation tools have become very popular because they target and streamline specific goals.

The goal of this particular visual programming language is to aid the user in designing and building web backends.

The project would involve building a frontend for the visual language and a backend platform that would run the language and user defined behaviour upon a trigger such as a HTTP request.

The hope is that it would allow someone to rapidly prototype and build a web service without the need of hosting their own infrastructure such as a web server or a database.

Acknowledgements

Many thanks to my supervisor Dr. Jason Quinlan for taking a student-proposed project and for always being patient and providing excellent feedback on my progress. I always wanted to do my final year project with him since he managed the Team Project module so well in my third year.

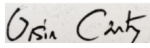
I'd also like to thank the committee members of Netsoc, both past-and-present. Netsoc is a society in UCC dedicated to technology and computer networking. I was a SysAdmin on the Netsoc committee for nearly 4 years. It gave me a chance to make life-long friends, manage a team of volunteers and build some amazing software used by loads of students in UCC (<https://github.com/UCCNetsoc>, <https://netsoc.co>, <https://netsoc.cloud>) The skills it allowed me to develop were invaluable in producing this final year project and I would have not been able to achieve this otherwise.

Declaration of Originality

Declaration of Originality

In signing this declaration, you are conforming, in writing, that the submitted work is entirely your own original work, except where clearly attributed otherwise, and that it has not been submitted partly or wholly for any other educational award. I hereby declare that:

- this is all my own work, unless clearly indicated otherwise, with full and proper accreditation;
- with respect to my own work: none of it has been submitted at any educational institution contributing in any way to an educational award;
- with respect to another's work: all text, diagrams, code, or ideas, whether verbatim, paraphrased or otherwise modified or adapted, have been duly attributed to the source in a scholarly manner, whether from books, papers, lecture notes or any other student's work, whether published or unpublished, electronically or in print.

Signed: 

Dated: 25th April 2022

Contents

Abstract

Acknowledgements

Declaration of Originality

List of Figures

Listings

1	Introduction	1
1.1	Background	1
1.2	Objective	1
1.3	Motivation	2
1.4	Report Outline	2
1.5	Artefact Summary	3
2	Analysis	4
2.1	VPL Representations	4
2.1.1	Dataflow Based	4
2.1.2	Drag-and-Drop Based	6
2.1.3	Diagram Based	7
2.2	VPL Runtimes	7
2.2.1	Tree-walking Interpreter	8
2.2.2	Bytecode Interpreter	8
2.2.3	Just-In-Time Compilation	8
2.2.4	Source-to-Source Compilation (Transpilation)	9
2.3	Structured Representations	9
2.4	VPL Perceptions	10

2.5	Requirements Analysis	11
2.5.1	Functional Requirements	12
2.5.2	Non-functional Requirements	12
3	Design	13
3.1	User Interface Service - <i>UISvc</i>	14
3.1.1	Choice of Framework - React	14
3.2	Language Design	16
3.2.1	Projects	16
3.2.2	Project Editor	17
3.2.3	Functions and Triggers	18
3.2.4	Statements	18
3.2.5	Blocks	19
3.2.6	Expressions	20
3.2.7	Branches	20
3.2.8	Namespaces	21
3.2.9	Type System	22
3.2.10	Handles	25
3.2.11	Models	25
3.2.12	Picker	27
3.2.13	Programming Language Interoperability	34
3.3	Language Library - <i>LangLib</i>	36
3.3.1	Project Representation	36
3.3.2	Language Runtime	36
3.3.3	Interpreter	37
3.3.4	Source-to-Source Compilation (Transpilation)	38
3.4	OpenFaaS	39
3.4.1	Docker / Open Container Initiative (OCI) images	39
3.4.2	OS-level Virtualization with Containers	40
3.4.3	Deployment	41
3.5	Docker Daemon	42
3.6	Docker Registry	42
3.6.1	Database - MinIO	42
3.7	Traefik	42
3.8	Projects Service - <i>ProjSvc</i>	43
3.8.1	<i>UISvc-ProjSvc</i> communication	43
3.8.2	Database - PostgreSQL	44
3.8.3	Project Management	45

3.8.4	Traefik Configuration	45
4	Implementation	46
4.1	Concepts Primer	46
4.1.1	<i>Head</i> and <i>Tail/Mut</i> structures	46
4.1.2	Wrapper Classes	47
4.1.3	Identifiers	47
4.2	<i>LangLib</i>	48
4.2.1	Type Representation	48
4.2.2	Namespaces	51
4.2.3	Defining Actions, Expressions and Triggers	52
4.2.4	Dimensions	55
4.2.5	Fuzzy Search with <i>n-grams</i>	56
4.2.6	Project Representation	57
4.2.7	Project Mutations with Unidirectional Data Flow (UDF)	62
4.2.8	Transpiler	64
4.2.9	Implementing Actions, Expressions and Triggers	67
4.2.10	Programming Language Interoperability	69
4.3	<i>ProjSvc</i>	70
4.3.1	JSON-RPC	70
4.3.2	Projects and Deployments	71
4.3.3	Improving Deployment Size and Time	73
4.3.4	Traefik Configuration	75
4.4	<i>UISvc</i>	77
4.4.1	Component Design	77
4.4.2	Editor State	77
4.4.3	React Contexts	80
4.4.4	Pannable and Zoomable Canvas	81
4.4.5	Recursive Components for Blocks, Expressions and Types	83
4.4.6	Layouts of Actions, Expressions and Triggers	86
4.4.7	Drag-and-Drop Support	87
4.4.8	Control Flow Wires	88
4.4.9	Picker	90
5	Evaluation	91
5.1	Open Day Feedback	91
5.2	HTTP Benchmarks	92
5.3	Deployment Stage Benchmarks	93
5.4	Requirements Checklist	95

5.5	Scalability Concerns and Future Work	96
5.5.1	Standard Library Additions	96
5.5.2	Tables and Queues	97
5.5.3	User Accounts	97
5.5.4	API Security	97
5.5.5	Real Time Editing	98
5.5.6	Visual Debugger	98
5.5.7	Instrumentation and Observability	98
5.5.8	Multi-Node Build Caching	99
5.5.9	Multi-Node OpenFaaS	99
5.5.10	Container Isolation Vulnerabilities	99
6	Conclusion	101
6.1	Reflection	101
6.2	Project Timeline	102

Bibliography

List of Figures

2.1	Unity Bolt (official image)	5
2.2	Blockly	6
2.3	DRAKON Language	7
3.1	System Architecture	13
3.2	Projects Landing Page	17
3.3	Infinite Canvas Graph and Project Pane	18
3.4	A statement invoking the <i>number::From Str</i> action	19
3.5	Variables defined in blocks	20
3.6	Expressions floating on the infinite canvas	21
3.7	A simple loop built using branches and the <i>If</i> action	22
3.8	A statement with two absent parameters of type <i>Num</i> and <i>Str</i>	24
3.9	A statement with an expression of absent parameter <i>b</i> of type <i>Num</i> and a mismatched <i>Num</i> expression supplied for a slot of type <i>Str</i>	25
3.10	A simple model definition of a response to a survey	26
3.11	Expressions for creating a new instance of the <i>SurveyResponse</i> model and referencing its fields	27
3.12	Create Function / Model layout	28
3.13	Floating statement bar	28
3.14	Insert Statement layout	29
3.15	Fuzzy search demonstrated	29
3.16	Expression Picker layout (all expressions)	30
3.17	Expression Picker layout (filtered expressions of type <i>Num</i>)	31
3.18	Expression with extension button	31
3.19	Extend Expression Picker layout	31
3.20	Expression extended with <i>Number to String</i> expression	32
3.21	Extend Expression Picker when a model instance is extended	32

3.22	Expression returning <i>author</i> field of <i>SurveyResponse</i> model instance in variable <i>response</i>	32
3.23	Create Variable / Model Field layout	33
3.24	Layout with well-formed type and name	33
3.25	<i>Array</i> expression where the automatically inferred type is <i>Array<Num></i>	33
3.26	Type Inference Override Picker	34
3.27	<i>Array</i> expression where the overridden type is <i>Array<Bool></i>	34
3.28	A Node.js code statement with a single captured variable <i>n</i>	35
3.29	Node.js dependency configuration	35
4.1	Linked-list representations of several types	48
4.2	Control-flow graph of a sample function	65
4.3	Control-flow-graph grouped into <i>sequences</i>	66
4.4	<i>UTypeRepr</i> component showing the deeply nested type for a variable	84
4.5	<i>UTypeRepr</i> recursion visualized in 3D	84
4.6	<i>+ Num</i> layout	87
4.7	Control Flow Wire <i>deadzones</i>	89
5.1	Fibonacci Calculator	92
6.1	Gantt Chart	102
6.2	WakaTime Graph	

Listings

3.1	A sample React counter component using hooks	15
4.1	Type linked list definitions.	48
4.2	Exerpt of type comparison with conversion and <i>Any</i> support. .	50
4.3	<i>NamespaceBuilder</i> in use defining a subset of the <i>Number</i> namespace	52
4.4	Dynamic use of <i>return</i> in the <i>Array</i> expression	55
4.5	The data structure representing a project.	58
4.6	Jump structure.	60
4.7	All possible <i>ProjActions</i>	63
4.8	<i>WProj</i> reducer	63
4.9	<i>From Str</i> action and <i>Plus</i> expression implementation	68
4.10	JSON-RPC Request Payload.	70
4.11	JSON-RPC Response Payload.	70
4.12	<i>ProjSvc</i> available RPCs.	71
4.13	Project build <i>Dockerfile</i>	73
4.14	Autogenerated Traefik configuration	76
4.15	<i>UIFloating</i> implementation	77
4.16	Editor state structures	78
4.17	Props definition of <i>UIExpr</i>	85
4.18	Layout helper methods in use with <i>exprLayout</i>	87

Chapter 1

Introduction

1.1 Background

Visual programming languages (*VPLs*) have existed for decades, varying in popularity and quality from year to year. VPLs are popular amongst beginner programmers but have failed thus far to capture the audience of professional programmers.

Despite these perceptions, both *workflow automation tooling* for enterprises and *visual scripting* for game development have risen in popularity over the past few years. These tools typically automate or glue actions together in response to events. They contain rudimentary control flow and the use of variables. Curiously, these tools remain popular yet share quite a lot in common with historically criticized visual programming languages.

1.2 Objective

Notably, the majority of successful workflow automation tools today are internet and cloud connected. Such examples include *AWS Step Functions* and *Microsoft Power Automate*. This suggests that there is an appetite for a modern, internet connected visual programming language. Therefore, the objective for this final year project aims to explore the **development of a modern, high performance visual programming platform for building a web backend**

1.3 Motivation

The primary motivation of the project is to ease the development of web backends.

Ordinarily, a user would need to host their own infrastructure when building a web application. A visual programming language combined with a platform for hosting and running the program would negate the need to do this.

Most programming languages are *general purpose*. For our needs, the visual programming language can be designed to be tightly integrated around its intended goal. Features specific to problems faced by web backends can be a part of the language directly. This would mean a user would not need to rely on some poorly maintained third-party packages for certain features. This is a much more user friendly and streamlined experience.

A visual programming language can potentially aid the user in more ways than a traditional *integrated development environment* (IDE). IDEs are still fundamentally built around code text files. A visual programming language can store the program as a *structured representation*. This means the editor can infer information about the program much easily. The editor can be bound to the language itself. The editor and language design can become one and the same. This means that editor design decisions can drive the design decisions of the language itself. This is an interesting aspect to explore as no current popular traditional programming language was built in this fashion.

1.4 Report Outline

The report covers the end-to-end development of a new visual programming language and platform for building web backends

Chapter 2 will provide a full breakdown and analysis of existing visual programming languages. Perceptions of these languages will be discussed. In addition to this, aspects from traditional programming languages that could be of use to a visual programming language will be explored. A requirements analysis specifying functional and non-functional requirements will be produced.

Chapter 3 will explore a high-level overview of a design to meet the requirements specified during the requirements analysis. High-level requirements

will be converted into low-level design requirements. All design decisions will be explored in further detail and elaborated on as necessary. Any third party frameworks and technologies used will be discussed to ensure context for a reader who is not familiar with them.

Chapter 4 will explore the implementations of the design choices in chapter 3. Internal aspects of the project will be discussed in detail. Chapter 5 will evaluate if the requirements defined in the analysis are met by the final design artefact. Chapter 6 will close the report with a short reflection of when and how the various aspects of the project were conducted.

1.5 Artefact Summary

This section provides a summary of the final constructed artefact.

The visual programming language was successfully implemented with features such as wire-based control flow, drag-and-drop expressions, a type system and model definitions. Editor features that allow a user to search for specific actions and expressions were included. In this fashion, the standard library of the language is self-documenting. HTTP requests can be handled and responses returned.

In addition to visual programming, a user can *inline* code of traditional programming languages such as JavaScript and modify variables in the visual programming language from the inlined code. The language feature scope can be extended vastly as importing third-party libraries is permitted.

A frontend was implemented with a fully interactive browser-based *Single-Page-Application*. A backend system was developed to store, build and deploy projects from the database.

A project is converted (*transpiled*) into the Go programming language for its speed and low resource usage. A project is securely isolated from other projects using operation system container features. When a project is deployed it can receive requests to its assigned subdomain and the behaviour defined by the user will be executed.

Technologies: *React, TypeScript, Node.js, Docker, containerd, OpenFaaS, PostgreSQL, Go, JSON-RPC, Traefik*

Chapter 2

Analysis

2.1 VPL Representations

This section offers an overview and categorization of different paradigms and representations in use by various visual programming languages (VPLs) today.

The positives and negatives of each paradigm are explored. Some paradigms have excelled in some fields where others have failed.

2.1.1 Dataflow Based

Dataflow or *flow graph* based visual programming languages represent the data path as a pipeline.

The program is typically represented as a *directed acyclic graph* of linked stages or operations. Each operation feeds its outputs into the next operation as inputs. The programmer is responsible for adding operations to the graph and linking outputs with inputs.

The primary advantage of this approach is the ability to *parallelize processing*.

A topological sort is performed on the graph to get a sequence of operations that need to be performed. If the runtime can confirm the next operation does not have dependencies on unfinished operations, it can run those stages in parallel.

The user can easily see all concurrent operations and communication across parallel operations is easily visible when outputs of multiple operations

converge together.

Traditional multi-threaded programming is complex and bug prone due to a reliance on synchronization primitives to prevent data races[36].

This particular VPL representation can eliminate most of these concerns.

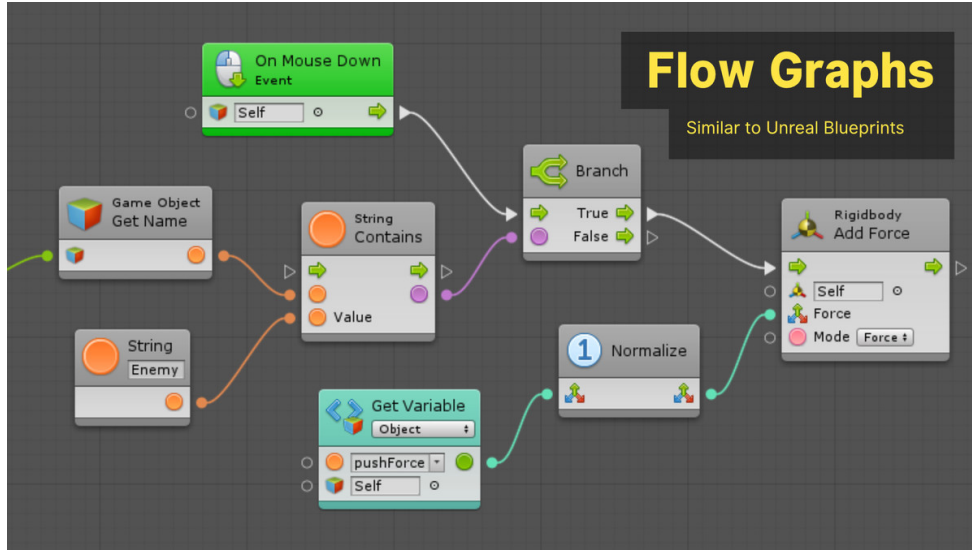


Figure 2.1: Unity Bolt (official image)

Figure 2.1 shows an example of such a VPL called Unity Bolt. Bolt is used for scripting with the Unity Game Engine. This particular block of code compares a string to see if a game object is an enemy. If it is and the *On Mouse Down* trigger is fired, the enemy will have a force vector applied to it.

This paradigm typically fails for a number of reasons, most often due to issues representing control flow. In the example above, control flow and data flow as paths in the same fashion. Execution begins at the *On Mouse Down* event and moves to the *Branch* operation. It is only here where the data flow dependencies are resolved and the calls to compare the string will be made. This can be very confusing to a new user unfamiliar with programming or using a bespoke tool.

Representing more complex control flow loops involves the same control flow output being executed multiple times. The flow of data does not appear necessarily "pure" in this case.

Another way this approach fails is "*visual spaghetti*"[33] Expressions when

chained together occupy large amounts of space and a large program is often difficult to decipher.

Allowing a user to manipulate and manage the path of graph edges between nodes is a solution to this. Unfortunately, this distracts from the goal of making programming easier and faster.

2.1.2 Drag-and-Drop Based

Drag-and-Drop based VPLs typically allow the user to construct a program from a set of predefined blocks and expressions. These blocks and expressions can be dragged and dropped together to implement functionality. This provides a level of familiarity to a professional programmer and a level of hand holding for a beginner programmer.

As a result, this approach allows expressions and statements to be easily built as a regular programming language. Execution is traditionally synchronous unlike dataflow based programming. However, this results in a simpler representation and does not suffer from the *visual spaghetti* problem.

Common criticisms of this approach relate to fatigue from drag and drop, to the lack of use of the keyboard.

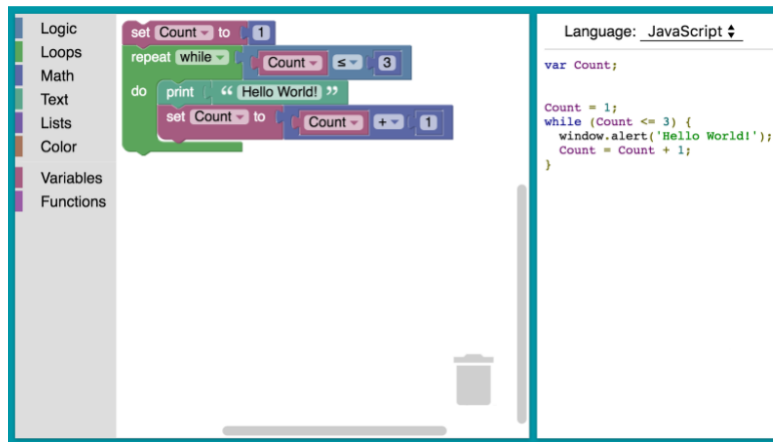


Figure 2.2: Blockly

Figure 2.2 shows an example of *blockly*, a web based visual programming tool by Google. The user can select from elements of various types from a

sidebar. There is no distinction between statements and expressions. The entire program relies exclusively on visual cues.

2.1.3 Diagram Based

Diagram based tooling such as DRAKON[90][12] allow a user to construct their program from decision trees. DRAKON was originally used to represent operations of the *Buran* Space Orbiter Program in a standardised fashion.

Most users will already be familiar with flow charts without the need to understand programming. DRAKON does not allow line intersections and lines can only take right angle turns. These constraints mean the decision diagram is probably the clearest VPL representation of program control flow.

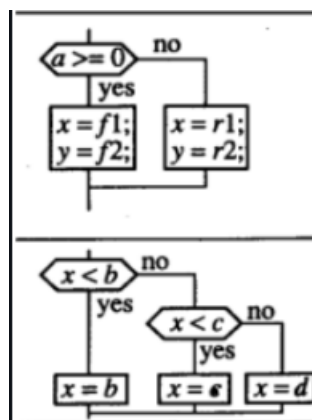


Figure 2.3: DRAKON Language

Figure 2.3 shows a DRAKON chart. Statements appear on the lines. DRAKON specifies different elements for functionality such as conditionals and switch statements.

2.2 VPL Runtimes

A runtime is responsible for executing the underlying functionality of the code written in a VPL. The underlying runtimes of visual programming languages are generally similar to the runtimes of traditional programming languages. Therefore, these elements were examined interchangeably.

2.2.1 Tree-walking Interpreter

An interpreter *interprets* the program instructions and executes a predefined block of interpreter code for each particular instruction. These interpreters are generally constructed as *virtual machines*. A register file, stack or heap is used to store values and objects during program execution.

Several interpreter architectures exist. Perhaps the most simple is a *tree-walking interpreter*[70]. This interpreter would walk the program graph executing each particular statement and following control flow. This is generally the slowest form of interpreter as large amounts of data must be read and processed for each individual step. Scratch uses a tree-walking JavaScript based interpreter called *scratch-vm*[72].

2.2.2 Bytecode Interpreter

An improvement to the previous architecture involves a transformation of the input code into a *bytecode form*. This is the architecture behind Unreal Engine Blueprint. It compiles to a custom VM bytecode for the UnrealScript VM[15][16]. As steps are now only small bytecode instructions, *spatial locality* is taken advantage of by the processor. The next instructions in the byte stream will be present in the cache of the processor. This means that instructions can quickly be read, executed and processed. The bytecode generator will normally decompose a higher-level operation into several small instructions.

The advantage of interpreter techniques is that generally execution can begin immediately. There is no intermediate build step. In the case of bytecode interpreters, the bytecode generation stage normally occurs during the parsing of the program.

The disadvantage is that interpreters generally are still slow enough to raise concerns about performance.

2.2.3 Just-In-Time Compilation

To speed up a bytecode based interpreter, *just-in-time compilation* (JIT) can be used. This is the approach used by the V8 JavaScript engine[31] (used in Google Chrome and Node.js) and the Java Virtual Machine[55] This involves analyzing *hot spots*, areas of code that see frequent execution during

interpretation and dynamically recompiling the *bytecode form* into *native machine code*. This native machine code is then ran directly rather than using the interpreter.

This particular technique offers massive speedups as the code being executed is near equivalent to compiled machine code output by a compiler.

Not all parts of the bytecode can be JIT compiled, falling back to the interpreter might be required if certain invariants about types of variables or branching cannot be confirmed.

Unfortunately, JITs are extremely complex to implement, CPU architecture dependent and vulnerable to exploits as raw machine code payloads are managed and executed[73].

2.2.4 Source-to-Source Compilation (Transpilation)

Source-to-source compilation or *transpilation* involves translating from one programming language to another. Two of the most popular transpiled languages used today are TypeScript and Dart which output JavaScript as their final build artefact.

The major advantage of transpilation is ability to leverage and make use of all of the features present in the target programming language. This can include performance optimizations, aspects of the standard library and the amount of third-party libraries available.

The major downside to transpilation is that it requires a build step to generate the code.

If the transpilation target language is a compiled language, an additional build step is required to build the generated code into a binary. This means that the time to build the binary is added onto the total build time.

If fast build times are required, it is vital that one chooses a target language with a fast compiler.

2.3 Structured Representations

In a traditional programming language, a project is typically stored as a directory of text files. These files are modified by the programmer in a text editor or IDE.

These files may or may not conform to the syntax of the programming language in use. They may have errors that prevent them from passing the first parsing stage of a compiler or interpreter. If code is in an unparseable, ambiguous state there is no way to mutate the program stored in that file automatically without human intervention.

Wright et al. (2013) explored at-scale automated refactoring of large C++ codebases at Google[91].

Their design involved a pipelined set of stages shown below:

1. Indexing - Textual form of code is passed to a compiler and an *abstract-syntax-tree* (AST) of the program is output and stored in a database
2. Transformations - Mutations are defined, matched and applied to the AST
3. Source Code Rewriting - Refactored code is generated from the AST and rewritten back to the original source file

If code in a visual programming language is stored as a *structured representation* (e.g. a control-flow graph), an implementation of this functionality would only require the transformation step.

No code indexing or rewriting is required. There is a guarantee that the program at rest is *always* in a readable, mutable state.

When this functionality is combined with a visual programming platform, it offers the ability of the platform to mass mutate all programs in a database. This could be done without the intervention of the project owner. This is an extremely powerful tool as it could be used to automatically patch vulnerabilities or perform migrations of deprecated functionality.

2.4 VPL Perceptions

It is well-known fact that programmers do not have a positive perception of visual programming tools. This section aims to aggregate the perceptions and defining opinions. To develop a VPL that will be used by both programmers and non-programmers alike, one will need to break the mould of these perceptions. Overarching themes were researched and observed. A summary of these is listed below.

Positives:

- Most syntax errors eliminated because syntax is enforced during program construction[46]
(e.g. placing an invalid statement in a particular block can be made impossible to do)
- Visual data schema design better than manual text-based schema design[45]
- Incredibly accessible, educational for both adults and children[45]
- Rapid prototyping, "*modern LEGO*"[46]

Negatives:

- Historical failures of *computer-aided software engineering* (CASE) and UML tooling[80]
- Vendor lock in, one is locked into the proprietary standard library and features of the language[45][80]
- Lack of *IntelliSense* functionality (hinting of class, function, variables names)[45]
- Lack of source control[45]
- Slow runtimes and generated code[80]
- Unextendible, programmers cannot easily add functionality to the language[80]
- Lack of a real type system and class definitions[45]
- Programmers can type faster than using visual elements to code[46]
- Seen as "*toys*"[46]

2.5 Requirements Analysis

Based on the data collected during the analysis, the following functional and non-functional requirements were elicited.

2.5.1 Functional Requirements

1. The platform must allow the user to create separate workspaces.
2. The language must be able to handle HTTP requests (query parameters, path matching, body) and respond as necessary.
3. The language must be optimized for coding rapidly and at speed.
4. The language must not be proprietary and force a user into a walled garden.
5. The language API must be easily extendable.
6. The language must feature an IntelliSense or hinting system.
7. The language must feature a type system comparable to a traditional programming language.
8. The language must feature a data definition system.
9. The language should make an attempt at preventing syntax errors.
10. The language should not confuse a user.

2.5.2 Non-functional Requirements

1. A project must execute securely alongside other projects
2. The runtime must allow a basic project to withstand a minimum of 500RPS (requests per second).
3. The runtime must allow a basic project to use less than 75MB of memory.
4. A basic project must deploy in less than a minute.

Chapter 3

Design

In this chapter we will explore the design of the system within the requirements defined by our analysis in Chapter 2

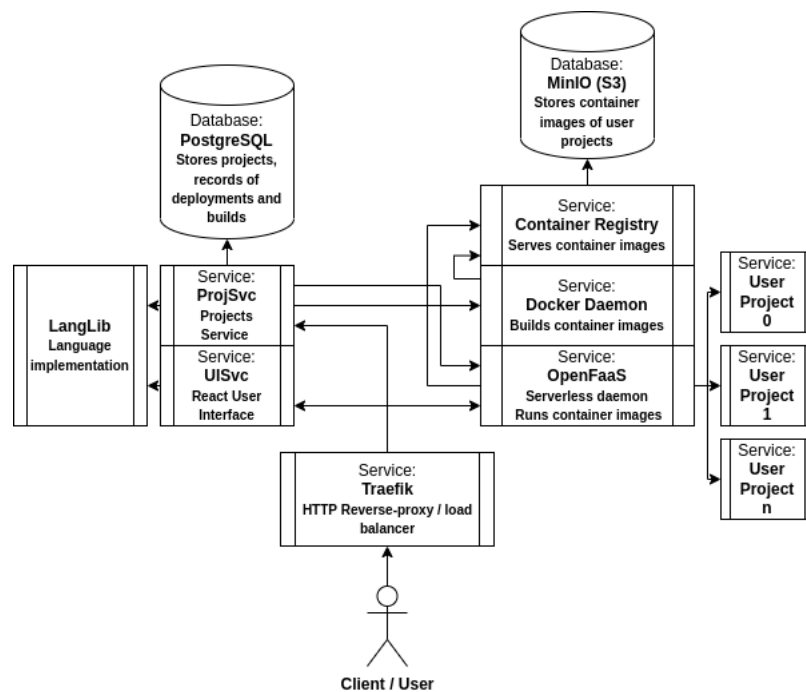


Figure 3.1: System Architecture

Figure 3.1 shows a high level overview of the final system architecture. Major system components are shown, both first-party (new, original code)

and third-party elements are present. Each component will be analyzed and their interrelationships with other components will be explored. All significant design decisions will be explored and justified.

3.1 User Interface Service - *UISvc*

When building a user interface there are two possible routes, a *native desktop application* or a *web application*.

In recent years with the advent of new browser-based JavaScript APIs[53], the web has become a powerful platform for building *complex, cross-platform, interactive* applications. A native desktop application would require the complexity of shipping binaries and dealing with OS-level primitives that are traditionally abstracted by the browser.

The frontend should be built as a browser-based, modern, *Single Page Application (SPA)*[52]. In a SPA, page reloads do not occur, rather the contents of the page are automatically updated by a scripting language such as JavaScript. Communication between the web page and backend typically occurs over JavaScript *fetch()* HTTP requests made to the backend web server. Making use of the application should be as simple as opening a web page.

3.1.1 Choice of Framework - React

There are a number of possible web frameworks to use when building a SPA. These frameworks streamline the process of managing complex components and property mutations in the *Document Object Model (DOM)*[51]. Currently the three most popular frameworks are *React*, *Vue* and *Angular*.

React was chosen as the web framework for the project.

By choosing React, we have the choice of using JavaScript or TypeScript as our implementation language. TypeScript was chosen for its superior ability at constructing large-scale applications[92]. It provides build time type-safety, API and type definitions, and great IDE support. These features are generally absent from JavaScript.

React allows one to create custom UI components that accept *props* (properties). These components are placed in a DOM-like hierarchy similar

to HTML or XML. A component consists of regular HTML, CSS and other components. Props are allowed to contain not just regular JavaScript objects but other components too. For example, when a component needs to have a dynamic child element inserted externally, the child component can be passed as a type of *ReactNode*.

When props or state within the component changes, React will automatically trigger a rerender of the component. This will update what the user sees on the page. React uses a tree-diffing algorithm to determine which components need to be rerendered when large state changes occur.

The primary reason React was chosen is because of its support for *functional components*. Vue 3 supports functional components but only gained maturity in early 2022[17]. Angular lacks support of any kind.

A functional component is a function that returns the element to be rendered when called.

Functional components massively simplify the development process over class based components because they are stateless. The majority of functional components in a typical React application are stateless rather than stateful and therefore the implementation is much cleaner and more performant.

Internal state in the component (that doesn't rely on the passed props) is accomplished via the use of *hooks*. This allows for updates to occur within the component in response to events that happen to the elements within.

The two most common hooks available are *useState* and *useEffect*. Listing 3.1 shows a demonstration of a simple counter element that uses two hooks. *useState* is used for stateful variables (the count) and *useEffect* can call a callback whenever updates of properties and stateful variables occur (the count update message).

Listing 3.1: A sample React counter component using hooks

```
1 interface CounterProps {  
2   initialValue: number  
3 }  
4  
5 function Counter(props: CounterProps) {  
6   const [count, setCount] = useState(props.initialValue);  
7  
8   useEffect(() => {  
9     console.log("Count was updated, count is now " + count)  
10  }, [count])  
11  
12   return (  
13     <div>
```

```

14     <span>{count}</span>
15     <button onClick={() => setCount(count + 1)}>
16         Increment
17     </button>
18 </div>
19 );
20 }
21
22 function App() {
23     return (
24         <div>
25             <h1>Counter</h1>
26             <Counter initialValue={0}/>
27         </div>
28     )
29 }

```

3.2 Language Design

The adopted programming paradigm for the language could be considered a *hybrid* of several of the approaches explored and analyzed in Chapter 2. The designed language has support for drag-and-drop expressions (from *drag-and-drop based* VPLs and path based control flow (from *decision-diagram based* VPLs).

Elements from both designs and traditional programming languages were considered and incorporated. This section will outline the decisions made and components of the language design.

The core concept behind the design decisions made for the visual programming language focus on **preventing ambiguity and showing a program as explicitly as possible**. Dataflow based VPLs often confused the user due the mix of the data flow and control flow. Approaches like this were generally avoided.

3.2.1 Projects

To allow for separation of concerns between workspaces, a user can create one or more *projects*. When creating a project, the user must provide a *unique identifier* for the project. This identifier is used to allocate a subdomain to the project. For example, if a user creates a project called *helloworld*, the *domain name* assigned for that project will be *helloworld.webpl.test*. When

the user is ready to deploy the project, the domain name will be used to forward matching requests to *functions* defined within the project.



Figure 3.2: Projects Landing Page

Figure 3.2 shows the design of the project landing page, a user can create, delete or select a project for editing.

3.2.2 Project Editor

In recent years, online web editors have become more and more popular. Examples include editors such as *Figma* and *Canva*. These editors make heavy use of the "infinite canvas" or *zooming user interface*[94], offering an infinitely pannable and zoomable canvas graph for editing. This reduces the need for context switching as the user is no longer required to move from pane to pane to use editing functionality. This functionality is also used by the data-flow based VPLs analyzed in Chapter 2. This was adopted in the project as it provides much more flexibility than is seen in the text based editors used for traditional programming languages.

Figure 3.3 shows the infinite canvas editor, a single *Hello World!* function is defined. The user can freely pan and zoom as they wish while editing. A *project pane* is overlaid, allowing the user to see any particular issues with the project they are editing and its deployment status

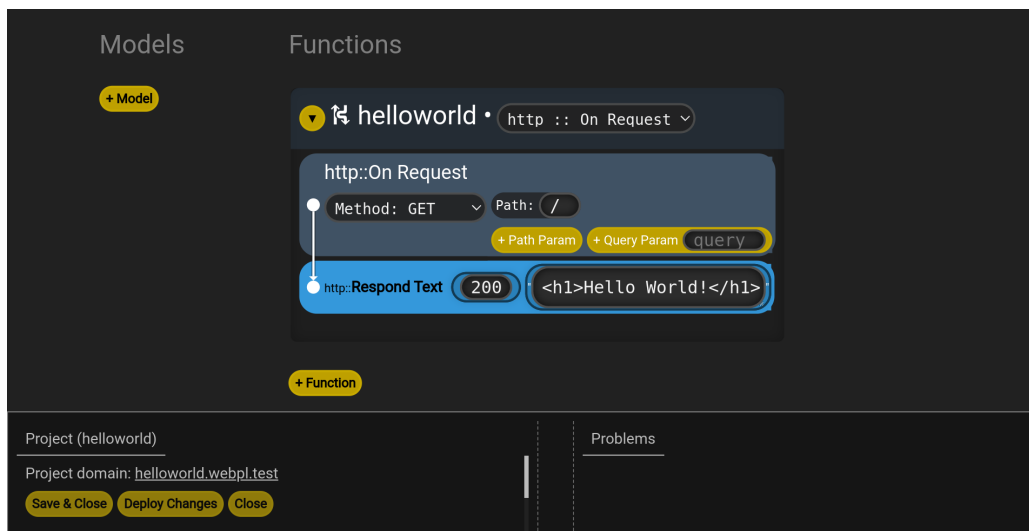


Figure 3.3: Infinite Canvas Graph and Project Pane

3.2.3 Functions and Triggers

A *function* contains a *block of statements* that is ran when the function is executed. A function is executed when a *trigger* occurs. A *trigger* is fired when the corresponding event described by the trigger's configuration occurs.

All functions a user implements must be backed by a trigger, this decision was made so a user is fully aware *how* and *what causes* their code to be executed.

One particular trigger is the *http::On Request* trigger, which fires when the project domain receives a *HTTP request*. The trigger can be configured to match particular request verbs, paths, query parameters and request bodies. When the trigger occurs, the user can extract the values of the matches.

3.2.4 Statements

Invocations of functions and behaviour in traditional programming languages typically return a value outlining a particular result of an operation. This is confusing and requires the callee to decide what control flow behaviour to perform next.

A *statement* is an invocation of an *action with a side effect*. An *action* takes a list of parameters (*expressions*), When the action occurs, the result

can cause the statement to *fallthrough* to the next statement or take one of many possible *branch paths*..

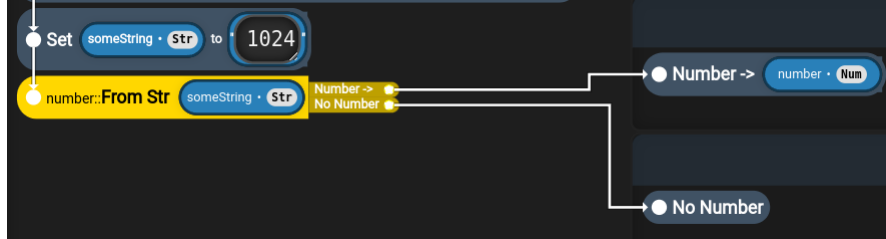


Figure 3.4: A statement invoking the *number::From Str* action

In this visual programming language, statements *are control flow* and take a particular branch *as the result of their invocation*. This design decision was made to make it explicitly clear to the user what the side-effect of an action is.

Each possible result branch can return a different set of parameters

Figure 3.4 shows an action converting a string to a number, two possible branches can occur. In the case of the string containing a number, control flow moves to the statement that stores the number in the *number* variable. In the case of the string not containing a number, control flow moves to the *No Number* statement which provides no return values.

3.2.5 Blocks

A *block* contains a list of statements and child blocks.

Many existing VPL implementations forgoe variable scopes, opting to make all variables global.

To avoid this messy anti-pattern, each block is considered a new *variable scope*. A user can define *local variables* in a block whose use is confined to that block and all of its descendant blocks

When execution passes into a child block from a parent block, variables defined in the child block are initialized and set to their default values. These variables are usable until execution returns to the parent block

Each variable is automatically added to the list of *expressions* the user can select from when coding a program.

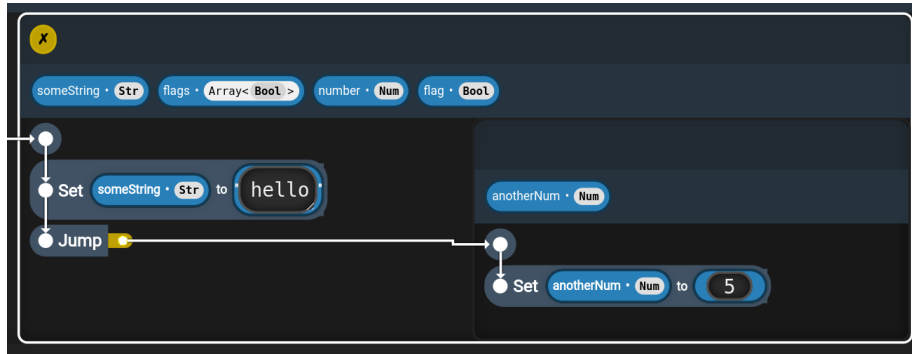


Figure 3.5: Variables defined in blocks

Figure 3.5 shows a block containing a child block. A statement is shown setting the *someString* variable to "hello" followed by a statement branching into the child block. The variables available for use in the child block would be the variables available in its parent blocks (*someString*, *flags*, *number*, *flag*) and the variables defined in its header (*anotherNum*)

3.2.6 Expressions

Similar to a traditional programming language, an *expression* takes a list of operands (more *expressions*). When invoked, the expressions returns a single value. The base expressions in the language are those that provide *literal* values (e.g. *true*, *false*, *0*, "hello") of the primitive types.

As shown in Figure 3.6, expressions float on the infinite canvas. A variable expression *someBool* of type *Bool* is shown.

Type-checking is automatically performed by highlighting slots of compatible types when dragging and dropping an expression. For example, when dragging an expression of type *Num*, any slots of *Str* (an incompatible type) will have their brightness lowered.

3.2.7 Branches

Traditional programming languages typically have *implicit* control flow at the end of blocks, e.g. automatically moving up to a previous block at the end of an *if* block.

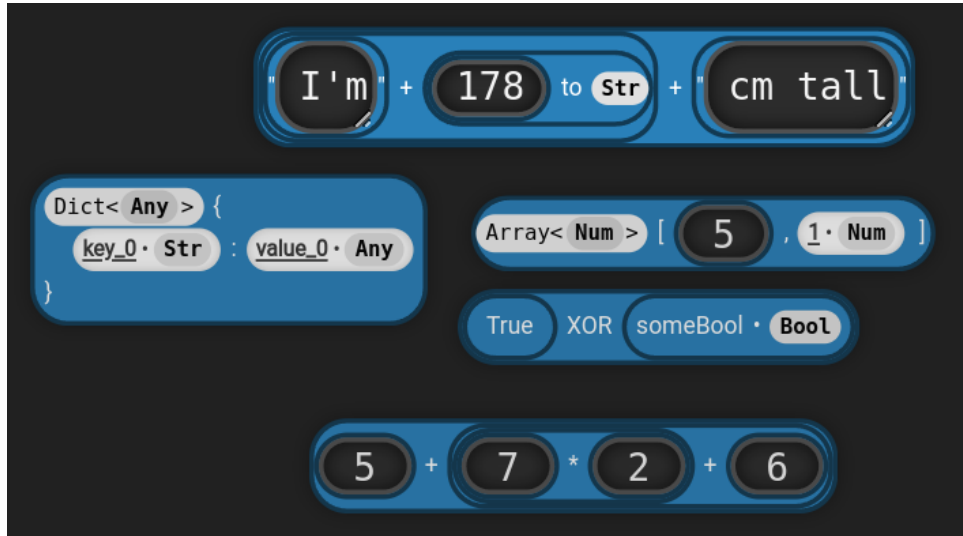


Figure 3.6: Expressions floating on the infinite canvas

In the visual programming language design, control flow is always *explicit*. This is a deliberate design choice to reduce user confusion.

Figure 3.7 shows how branches support one of 3 forms:

1. Branching into a child block of the current block
2. Branching within the current block
3. Branching into the parent block of the current block

Unreachable statements have their brightness reduced to highlight to a user that execution can never reach that path.

3.2.8 Namespaces

A *namespace* contains a logical grouping of actions and expressions. A commonly seen namespace in the language is the *Number* namespace which contains expressions that implement all numeric operations (e.g. +, -, *, /) and actions that perform operations such as parsing strings for numbers.

Namespaces are present in many traditional programming languages such as TypeScript and C++. Their primary goal is to reduce naming conflicts.



Figure 3.7: A simple loop built using branches and the *If* action

Namespaces are typically postfixed with `::`, e.g. `http::` for the HTTP namespace

Each project has its own namespace (of the same name) for definitions within the project

3.2.9 Type System

The type system was deliberately chosen to be similar to the type system of *Javascript Object Notation*[77] (JSON). This was done because JSON is one of the most common data exchange formats and many users will be already familiar with it.[74] As JSON support is near universal, this decision offers forward compatibility as a type system like this will have excellent interoperability with any other existing programming language if required. The number of additional types in the type system is deliberately chosen to be small to reduce confusion

Type Structure

A *type* can take 1 of 2 forms:

1. A *final* type
2. A *container* type

Final Type Categories

Final types must be one of the following categories:

1. 64-bit IEEE 754[34] double-precision floating-point number (*Num*)
2. string (*Str*)
3. boolean (*Bool*)
4. a model identifier
5. a handle identifier

JavaScript similarly makes use of IEEE 754 double-precision floats for numeric types[14]. This offers a user a safe integer range (the range of integers where no integer will suffer a loss of precision) of $-2^{53} - 1 \dots 2^{53} - 1$

Container Type Categories

Container types must be one of the following categories

1. dictionary (*Dict*)
2. array (*Array*)
3. a reference to a variable (*Var*, the child type specifies the variable type)

A container type *contains* another container type or a final type. This is typically represented by $<$ and $>$ enclosing the child type.

For example:

A dictionary of arrays containing booleans: *Dict<Array<Bool>>*

A variable reference that contains a boolean value: *Var<Bool>*

A variable reference should not be able to contain a variable reference.

Type Conversions

There is a need for *type conversions* in the type system. Many languages have *implicit type conversions*. These are conversions that are not automatically specified by the user.

For example in *C++*, any positive integer literal can be implicitly convertible to a positive boolean value. The type system in this visual programming language only has a single implicit conversion. Any variable reference (type *Var*) is implicitly convertible to its value type. For example, *Var<Bool>* can be passed to any parameter slot of type *Bool*. This conversion exists to allow the values stored in variables to be used in actions and expressions.

All other conversions should be *explicit type conversions* that make use of an expression to perform the conversion. For example, a *Num To Str* expression which takes a single parameter of type *Num* should be used to convert a number to a string.

Type Checking

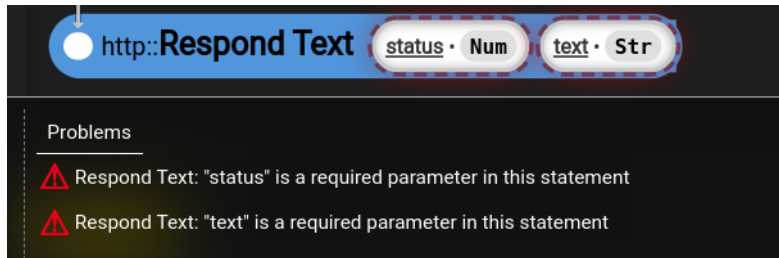


Figure 3.8: A statement with two absent parameters of type *Num* and *Str*

Type checking should be performed on all parameter slots of statements and expressions. This should be done by traversing the program and checking each slot.

The type checking algorithm should check if the type returned by the expression in a slot matches the type the slot expects.

The user should be provided with an aggregated list of problems in the project before they are allowed to deploy it.

A traditional programming language has errors output by the build tool. These errors typically reference line numbers. Editors and IDEs will highlight the entire line causing the problem.

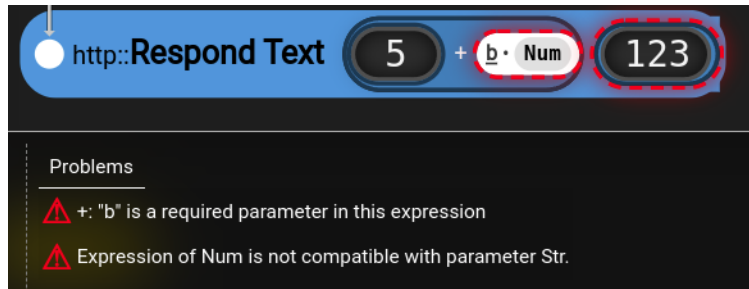


Figure 3.9: A statement with an expression of absent parameter b of type Num and a mismatched Num expression supplied for a slot of type Str

For the equivalent functionality in the visual programming language, only highlighting a line was not specific enough. The slot causing the error should be highlighted directly. Figure 3.8 and Figure 3.9 show two scenarios where type checking is finding and highlighting errors.

3.2.10 Handles

A *handle* is type that represents access to a specific resource. Handles are *opaque* in the fashion that the user cannot access the data a handle represents directly.

Handles can only be passed to actions which then perform operations or return data related to the resource the handle represents.

A user cannot define their own types of handles. They exist to represent serverside resources with a fully private and encapsulated underlying representation while offering a public API for the user to manipulate the resource. This decision is deliberate as it allows updates and changes to the underlying representation without interfering with the public API a project might depend on.

Handles were inspired by their use in the core APIs of the Microsoft Windows operating systems. Their use in the *Win32 Programming APIs*[43] accomplishes the same goals as their use in our language.

3.2.11 Models

A *model* allows a user to define a named data structure with any number of ordered *fields*.

A *field* can be of any type except the type of a *variable reference* or *handle identifier*. This means that models are allowed to contain other models.

When a new model is created, the model type is automatically added to usable types for variables and parameter slots.

Models can be serialized to data exchange formats such as JSON and the serialization keys are customizable. Figure 3.10 shows a model definition, each field appears in the order it was added.

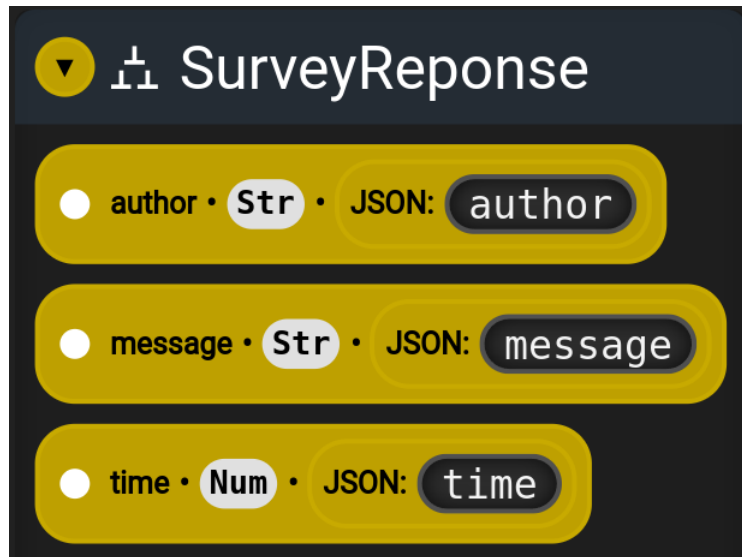


Figure 3.10: A simple model definition of a response to a survey

Additional expressions are automatically added that allow the user to instantiate a model and reference its fields. Figure 3.11 shows an example of the expressions added for the *SurveyResponse* model shown in Figure 3.10. An expression for instantiating the model contains a parameter slot for each field in the model. A variable called *response* storing an instance of the *SurveyResponse* model is also shown

A model field can be *archived*. This means the field is no longer in use and deprecated. When a model is instantiated, all of its fields are set to default values. This allows for archived fields to remain on the model but if no value is supplied for the field it will simply be set to the default value. This means that old code can continue to reference the field, but new code is not required to.

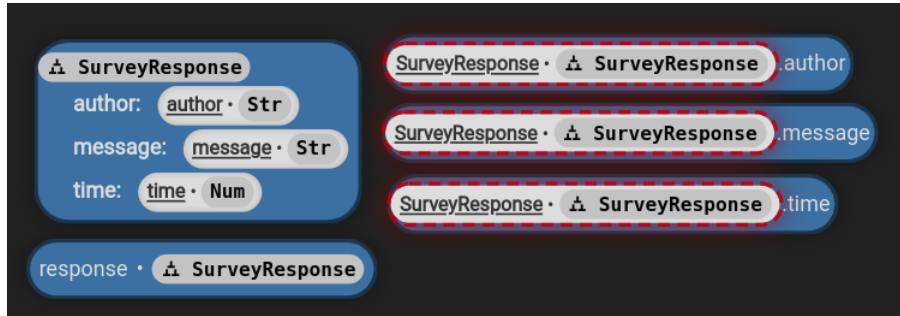


Figure 3.11: Expressions for creating a new instance of the *SurveyResponse* model and referencing its fields

This behaviour is inspired by *Google Protocol Buffers*[30].

Google found that making fields on a data structure optional and thus having default values offered excellent forward and backwards compatibility. Fields can be added or marked as unused (*archived*) at will without breaking existing code. This feature helped Google dramatically reduce the amount of downtime and surprise issues occurring in production[22].

The user is not allowed to specify the default value. Default values for a type such as *Num* would 0, *Str* would be an empty string and *Bool* would be *False*.

3.2.12 Picker

The *Picker* is the core UI component the user uses to insert visual elements in the language. When examining existing visual programming languages in Chapter 2, many drag and drop languages were found to require the user to drag items from a pane or sidebar. This increases the amount of mouse travel time the user spends building expressions and statements.

Offering a UI component that is always close to the users mouse is an important feature.

The *Picker* is a dialog that is overlaid on the project canvas. During normal operation it is not visible, Many input actions can cause the *Picker* to appear. When the *Picker* appears, it will always appear under the users mouse. A specific input action can cause the *Picker* to appear in a specific layout. Thus, the *Picker* is multi-faceted and serves many different functions for inserting elements into the currently edited project.

This section will outline the functionality and layouts the *Picker* can assume.

Create Function / Model

Creating a new function or model is as simple as selecting the *+ Function* or *+ Model* on the project canvas. This opens up the *Picker* on the mouse location ready to input the new function or model name. This can be seen in Figure 3.12.



Figure 3.12: Create Function / Model layout

Insert Statement

The *Insert Statement* layout is triggered from a *floating statement bar* (seen in Figure 3.13 that can appear between existing statements when hovered over. This bar provides many different possible buttons that will perform changes relative to the position of the bar.

The *Insert Statement* Picker layout can be opened by pressing *+ Action* button

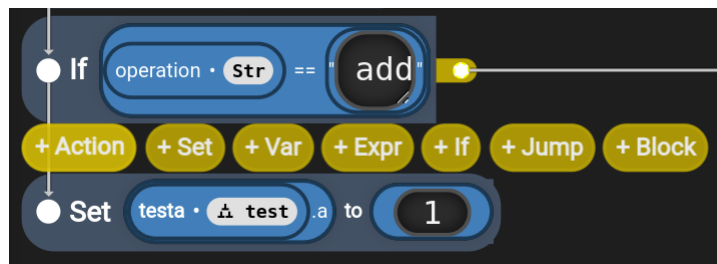


Figure 3.13: Floating statement bar

The layout provides a list of all actions that the user can use for their statement. It provides tabs to filter by the namespace an action is a member

of and *fuzzy text search* for searching for actions by their name. This design decision was made to prevent the user from having to consult documentation. Every single possible action they can use within a program is easily visible and searchable. Many editors and IDEs provide *IntelliSense*[40] which hints functions, variables and properties within a library. However IntelliSense typically does not have fuzzy search nor is a first-class feature of the editor. The visual programming language provides this feature *as the way of programming* rather than simply acting as an *assist to programming*.

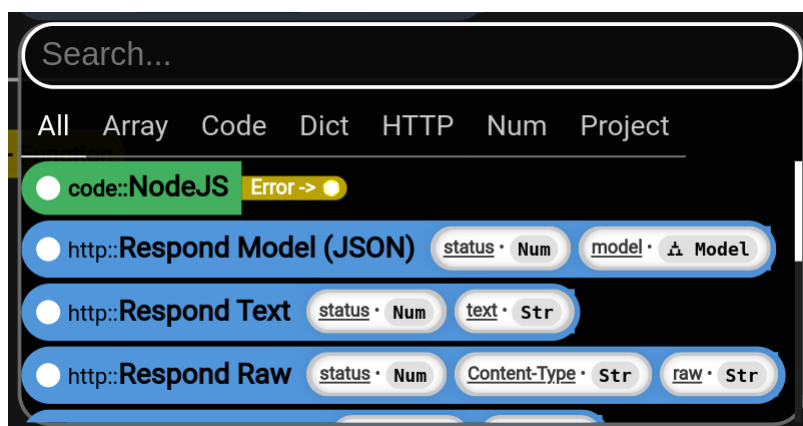


Figure 3.14: Insert Statement layout

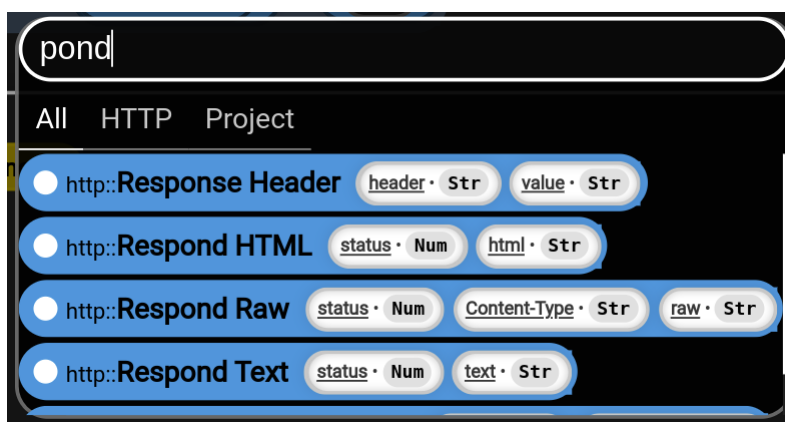


Figure 3.15: Fuzzy search demonstrated

Insert Expression

There are two ways to trigger the *Insert Expression* Picker layout. One is via the *+ Expr* button on the *floating statement bar* mentioned previously. In this case, the user can select from any expression available (shown in Figure 3.16). The other way to trigger the layout is via clicking on an existing parameter slot. This will open the layout but filter any expression that does not match the type of that parameter slot. For example, one can select a parameter slot of type *Num* and the layout will appear only showing expressions that return type *Num* (shown in Figure 3.17). This design decision was made as it allows the user to quickly get access to the expressions they are looking for rather than wasting time searching.

Similar to the *Insert Statement* layout, namespace tab filtering and fuzzy search is also available. Model expressions and variable expressions in the current block scope are accessible within the *Project* tab.

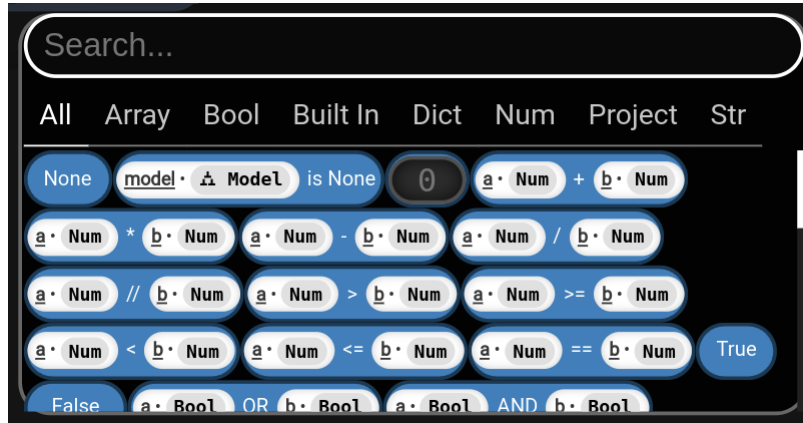


Figure 3.16: Expression Picker layout (all expressions)

Extend Expression

The *Extend Expression* Picker layout is used to add an existing expression into the parameter slot of a new expression. This is done to save the user the time they would have spent adding a new expression and manually dragging and dropping the existing expression into a parameter slot. When an expression is selected (as shown in Figure 3.18) a button can be pressed to extend the

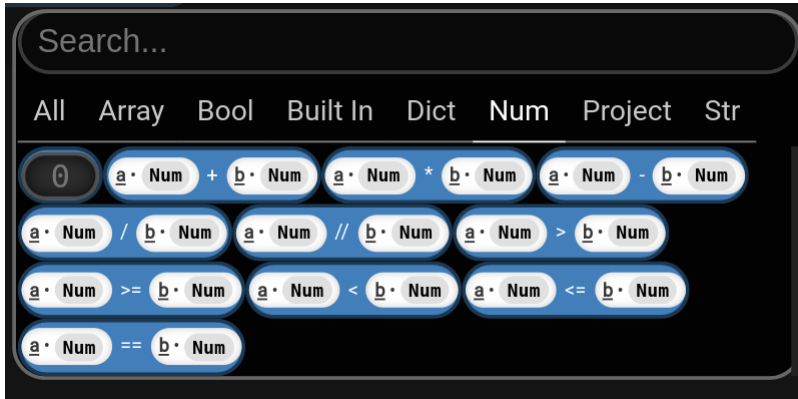


Figure 3.17: Expression Picker layout (filtered expressions of type *Num*)

expression. This will open the layout where a new expression can be selected (Figure 3.19).

For example if the expression in Figure 3.18 is extended with the *Number* to *String* expression, the resulting expression can be seen in Figure 3.20

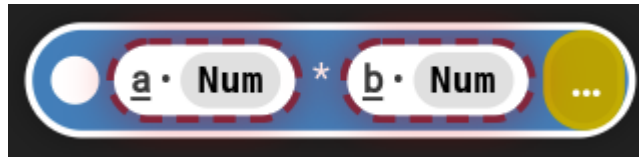


Figure 3.18: Expression with extension button

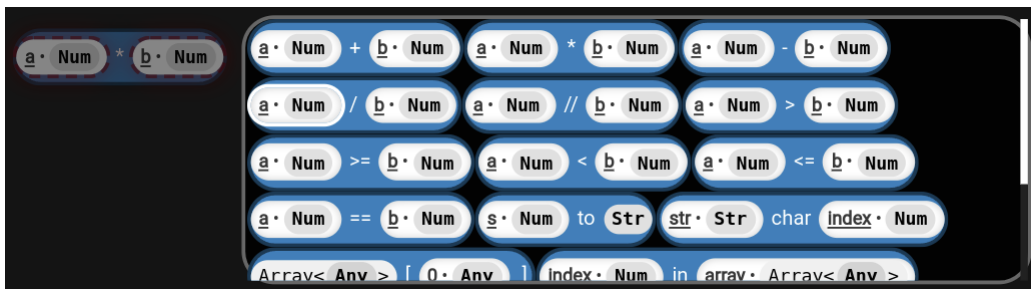


Figure 3.19: Extend Expression Picker layout

Expression extension is also used to select the fields of model instances. Figure 3.21 shows the extension of a variable containing an instance of the *SurveyResponse* model mentioned previously. Fields of the *SurveyResponse*

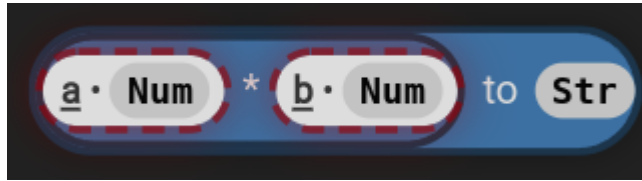


Figure 3.20: Expression extended with *Number to String* expression

model can be selected in the Picker. For example, Figure 3.22 shows if the *author* field was selected. This new expression will then return the value stored in the *author* field of the model instance in the variable *response*.

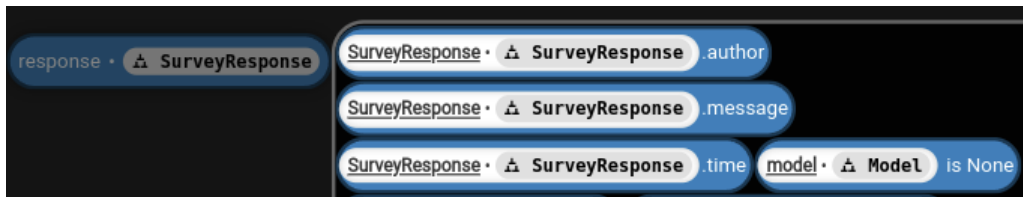


Figure 3.21: Extend Expression Picker when a model instance is extended



Figure 3.22: Expression returning *author* field of *SurveyResponse* model instance in variable *response*

Create Variable / Model Field

For the *Create Variable / Model Field* layout, a name field and type picker is provided (shown in Figure 3.23), this allows one to build a compound type if using container types such as *Dict* or *Array* or simply make use of a final type like *Num* or *Str*. In addition to this, model and handle types are also present. Fuzzy text search of types is provided as expected from other layout designs.



Figure 3.23: Create Variable / Model Field layout

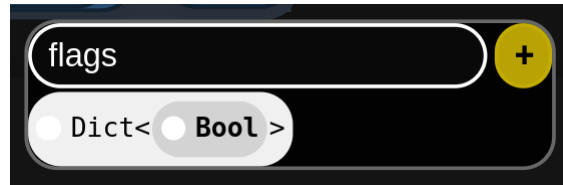


Figure 3.24: Layout with well-formed type and name

Type Inference Override

Some expressions use *local type inference* (automated deduction of a type) in determining its return type. An example of such an expression is the *Array* expression. The *Array* expression deduces the type of the array by examining the types of the parameters passed to it.

For example, if the types passed to the expression are *Num*, the array expression will return the type *Array<Num>* (shown in Figure 3.25). In some cases this is not desired behaviour, particularly with the use of the *Array* and *Dict* expressions. The user may want to override the type an array is expecting.



Figure 3.25: *Array* expression where the automatically inferred type is *Array<Num>*

By clicking the type on the expression, the *Type Inference Override* Picker layout (shown in Figure 3.26) opens, allowing one to search and select for a

type.

Figure 3.27 shows the array type overridden to return *Array<Bool>*



Figure 3.26: Type Inference Override Picker

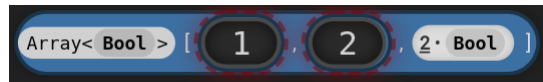


Figure 3.27: *Array* expression where the overridden type is *Array<Bool>*

3.2.13 Programming Language Interoperability

As discussed in the analysis in chapter 2, a common criticism of visual programming languages is that they are *proprietary*. Programmers shun these tools because they feel limited by what they can achieve.

To avoid this from occurring in this visual programming language, it offers the ability to also program in traditional programming languages. A user should be able to inline code in a well-known, established programming language such as *Node.js*. They should be able to modify existing variables in the visual programming language from the traditional programming language. This should be achieved by offering the user the use of *capture lists*. This is inspired by *lambda captures*[2] used by the C++ programming language. In a C++ lambda (anonymous closure) function, a user can pass a list of variables by *reference* or *value* to make the variables usable inside the function body. Similarly, in the design of this visual programming language, the user should be offered the ability to map variable names inside the traditional programming to variables in the visual programming language. When execution of the code in the traditional programming language completes, the variables mapped

previously have their values updated if they were modified by the traditional programming language.

Figure 3.28 shows a *Node.js* statement capturing the variable n . The Node.js code being run assigns n to a value returned by a call to `Math.pow`. When the statement is finished executing, The variable n in the visual programming language will be updated to the value assigned during the execution of the Node.js code.

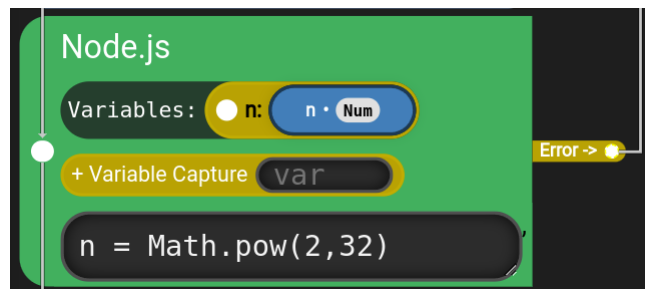


Figure 3.28: A Node.js code statement with a single captured variable n

As explained previously in this chapter, the type system is based on the widely supported serialization format JSON. This means that traditional programming languages will be able to easily handle any variables captured by these statements.

Different programming languages should be configured as *environments*, a user should be able to specify the package configuration for that particular programming language. Figure 3.29 shows the package configuration for Node.js, one dependency *lodash* is shown.

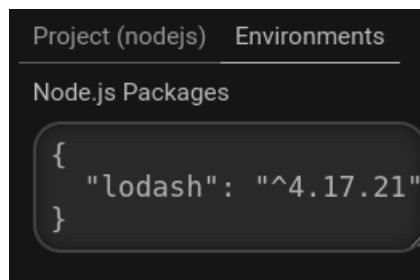


Figure 3.29: Node.js dependency configuration

3.3 Language Library - *LangLib*

There is a need for code processing and manipulating data structures representing the visual programming language to be shared between the frontend *UISvc* and the backend *ProjSvc* (*discussed later*).

This is accomplished by *LangLib*. *LangLib* is used by the frontend to provide editing functionality and used by the backend to build a project into its final runnable artefact. As *LangLib* is required on the frontend, it must be implemented in the same programming language and environment discussed previously (TypeScript).

3.3.1 Project Representation

LangLib offers transformation functionality similar to the refactoring operations explored in section 2.3.

A project is stored as a structured representation in various data structures (discussed in Chapter 4). This means a project is *always* in a valid state. A project may have errors that need to be resolved before it can be successfully built, but the project is always readable and mutable by the functions provided by *LangLib* to its consumers.

LangLib functions provide the means to edit a project exclusively via defined transformations. For example, a transformation could create a new statement in a block. There is no other way to modify a project. This allows for all modifications to a project to be logged and observed.

3.3.2 Language Runtime

There are a number of design options when determining how to execute the language at runtime. Some of these steps involve preprocessing which typically means there is an intermediate build stage. Two possible options were explored, an *interpreter* and *source-to-source compilation (transpilation)*

When exploring the design of the runtime it must meet the requirements analysis explored in Chapter 2 and must be suitable to implement the behaviour of the programming language design defined earlier in this chapter.

A summary of the runtime design priorities:

1. Fast build speed (if a build step is required)

It is expected that a user will be continually modifying a project and rebuilding it to test their behaviour. Build speeds of any artefact must feel quick and responsive to the user.

2. Fast bootup speed

This is required because a users project can have a *cold-start*. If the project is not yet running, the project must be initialized and a request cannot be processed until the project is ready.

The runtime should be able to quickly initialize the project and begin execution.

3. Fast execution speeds

A project should be able to handle many simultaenous requests, as a result execution speed should be a priority.

4. Low memory usage

As multiple projects can be deployed simultaenously, the memory footprint of each project should be small within the runtime.

5. Garbage collection

As the language design does not require the user to manually manage memory allocations, the runtime must offer a garbage collector. The garbage collector should be responsible for freeing objects which are no longer referenced

6. Standard and 3rd-party library support for web backend features

The runtime design must be implemented in a language that has good support for implementing features related to web-backends. This support can come in the form of being offered by the standard library of the language or by 3rd-party libraries available for that language.

3.3.3 Interpreter

An initial tree-walking interpreter was explored in the design phase but abandoned due to the prospects of low performance and complexity of implementation required to improve performance.

3.3.4 Source-to-Source Compilation (Transpilation)

Transpilation is the primary focus for the runtime implementation of the language. The main decision made in this chapter was determining which target programming language to transpile to.

Pereira et al (2017) explored runtime, memory usage and energy consumption of twenty seven well-known programming languages[64]. The results from this paper were used as a benchmark. Programming languages that offer a garbage collector and that were commonly used for building web backends were chosen to be examined for the design.

Languages such as Python and Java rely on external application servers such as Gnuicorn and Apache Tomcat for handling HTTP requests at scale. Often, these application servers are bloated, and are slow to boot up. Tomcat often has startup times of 10 to 60 seconds[82]. This startup time is unacceptable. Keeping the number of external dependencies needed to implement core functionality low is of high priority.

Go and Node.js were both of interest because they offer built-in HTTP server functionality[29][62] as part of their standard library.

In the paper results, Go scored in 14/27 in energy efficiency, 7/27 in speed, and 2/27 for memory usage. Node.js scored 17/27 for energy efficiency, 14/27 for speed and 20/27 for memory usage. In HTTP request handling benchmarks, Go routinely outperforms Node.js[76]

These differences can be explained by the architecture of each language. Go is a concurrent language built around the use of *goroutines*. A *goroutine* is a lightweight thread of execution with a small stack. During execution, many *goroutines* are scheduled across many operating system threads by the Go scheduler[71].

Typically this means that each HTTP request to a Go based web server would create a new *goroutine* to handle that request.

This differs from Node.js, which uses a single thread of execution and an event loop[63]. This execution model allows for asynchronous I/O to be performed. The event loop starts by processing any pending timers or callbacks. This is followed by checking for if any polling I/O has returned. If I/O has returned, any callbacks waiting on that I/O will be called on the next iteration of the loop.

I/O is performed by multiple threads, but JavaScript execution is limited

to a single thread. To take advantage of multiple host processors, multiple processes need to be ran which each consume large amounts of memory.

This leads to the choice of Go as the programming language to use as the transpilation target as Node.js was not felt to be suitable enough.

It should be noted that as Go is a compiled language, so an extra build step is required. However, Go is well known for its fast build speed.[6] which will add only a neglibl delay to the deploy step the user experiences.

Implementation of the transpiler will be discussed in Chapter 4.

3.4 OpenFaaS

A project should have the ability to execute on demand when a trigger fires. In addition to this, an executing project should not be able to interfere with the resources or execution of other projects. A project should never be able to take control of the host system.

To accomplish this, *OpenFaaS (OpenFunctions-as-a-Service)*[60] is used.

OpenFaaS is an open-source serverless engine. OpenFaaS exposes an API where serverless functions can be created, Once a serverless function has been created, it can be executed by sending a request to the OpenFaaS API.

When the function receives a request, OpenFaaS will boot the function if required, wait for the function to be ready and then pass the request through to the function. When the function has stopped receiving requests after a period of time, OpenFaaS will shut down the function to save resources.

OpenFaaS isolates functions by running them in *Docker or Open Container Initiative (OCI)*[8] containers. A *container* is an OS-level virtualization and isolation technique built using features offered by the Linux kernel.

3.4.1 Docker / Open Container Initiative (OCI) images

A running container is generally an instance of a Docker / OCI *image*[57]. An image typically contains a filesystem composed of many layers. Each of these layers inherit the layer above. For example, a base image can be an *ubuntu* image, containing the files found in a base installation of the Ubuntu Server OS. One can then inherit from the *ubuntu* image layer and add in

their own custom resources and binaries in the next layer. One or more layers are applied on top of each other to create a complete filesystem[58]. JSON metadata is attached to each layer specifying associated resource requirements and configuration. For example, the final layer will typically specify what process to run when an instance of that image is ran as a container.

Generally images are built via *Dockerfiles*[9]. A *Dockerfile* specifies a set of build steps and Linux commands to build an image layer. An image is built by providing the *Dockerfile* to a *Docker daemon* which performs the build in several intermediate containers.

Images are pushed and stored in *image registries*. The most commonly used image registry today is the Docker Hub registry[11] which offers images of commonly deployed backend software like the *nginx* and *Apache* web servers.

OpenFaaS expects that each serverless function has its resources and binaries stored in an OCI-compliant container image.

3.4.2 OS-level Virtualization with Containers

Creating a container invokes a process called *containerd* which pulls the image from its origin image registry. *containerd* then invokes *runc*[56]. *runc* is responsible for setting up the container process. It does this by creating the process within Linux *namespaces*[23].

When a process is within a Linux namespace, it is isolated from the particular resource the namespace represents. For example, if a process is in a *PID (Process Identifier)* namespace, it can only see itself or its child processes. It is not aware of any other processes running on the machine unless they are also present in the same namespace.

Generally a process will be placed in a new *filesystem mount* namespace containing a filesystem mount with the contents of the container image.

In addition to aforementioned resources, namespaces exist for many other kernel resources[24] such as *user accounts*, *network adapters*, *IPC (inter-process communication)*, *UTS (time configuration)* and more. This is where the concept of OS level virtualization comes into play. Fundamentally, the process in the container is unaware that it is sharing the system with many other containers.

To limit resource *usage*, *cgroups*[25] or *control groups* are used. These allow specific resource allocations to be assigned to groups of processes. Limitations

can be placed on many different allocations such as CPU time, memory or block device (I/O) usage.

Linux security *capabilities*[26] are used to make the container process unprivileged. This limits the number of *system calls* the process has access to. This prevents the container from damaging or making changes to the host system. For example, the capability `CAP_SYS_ADMIN` is not permitted within the container. `CAP_SYS_ADMIN` guards several aspects of critical system functionality. One such functionality is the ability to use the *setns()* system call. If a container process had access to *setns()* it could use it to escape any namespaces it is within and take control of the host system.

While *capabilities* provide general control over certain kernel functionality Linux *seccomp* (*Secure Computing*)[27] provides control over specific listed system calls. The default seccomp profile[59] prohibits various system calls that can cause security issues when used by processes inside the container.

3.4.3 Deployment

OpenFaaS can be deployed in two scenarios. It can be deployed on a Kubernetes cluster as a collection of containers or alternatively can be deployed as a system called *faasd*[61]. *faasd* does not require a clustering mechanism and can simply be installed on a server or developer machine for testing.

faasd was chosen because of its support for the Linux kernel *cgroup freezer*[79]. When a function is not in use, *faasd* uses *runc*'s *freeze* command which freezes the *cgroup* the container processes are a member of. This will pause/suspend the processes and if memory pressure occurs, the process memory will be swapped out by the kernel automatically. The function is labeled as having 0 replicas during this period.

When a request comes in, *faasd* can automatically *resume* the container. This unfreezes the processes and is generally faster than a cold-start.

The Kubernetes deployment of OpenFaaS was avoided because its *scale-to-zero* support is only offered by a paid addon. In addition to this, the time to scale from zero can be anywhere from 1-5 seconds as it does not make use of the *cgroup freezer* functionality. *faasd* unfreeze was found to occur with the process ready to handle a request with an average delay of approximately 0.2 seconds

3.5 Docker Daemon

A Docker *daemon* is deployed as it is used to build container images.

3.6 Docker Registry

A Docker *Registry* is deployed as a service, this registry exists to store and serve any container images that are built and deployed onto OpenFaaS.

A remote registry (e.g. Docker Hub, as mentioned earlier) is not used because of latency and security concerns.

3.6.1 Database - MinIO

The Docker Registry stores each container image as a record in an AWS S3 compatible object store. MinIO[44], an open-source object store with an AWS S3 compatible API is used rather than relying on AWS S3. This design decision was made to ensure data is stored locally rather than relying on an external cloud system.

3.7 Traefik

Traefik is a HTTP reverse proxy and load-balancer. It sits in front of the services used by the project and reverse proxies connections into them.

The primary reason Traefik was chosen was for its excellent support for dynamic reconfiguration[83]. Configuration can be *discovered* at runtime rather than exclusively loaded from a static config file.

Traditional web servers and reverse proxies like *nginx* and *Apache* are built around textual configuration files. If a change is made at runtime, the config file needs to be modified and a configuration reload signal sent to the server process[18]. This is cumbersome to implement and thus a traditional web server was avoided.

Traefik discovers configuration via *providers*. Many providers are supported. Traefik can pull configuration from key value stores like *Redis*, *Hashicorp Consul*, *Apache ZooKeeper*, *etcd* and more[83].

The particular provider chosen for use in the FYP was the *HTTP* provider[85]. When a HTTP provider is in use, Traefik will continually poll a HTTP URL at a set interval. The HTTP URL should return the configuration that Traefik is to use. Traefik will *diff* its current configuration against the returned configuration and implement the configuration changes without halting any in progress requests.

This behaviour is excellent as it allows us to continually update the web server configuration as users create, modify and delete projects. Web server configuration can be automatically updated with minimal downtime.

For SSL/TLS support, traditional web servers would require one to manually maintain a directory of certificates and maintain configuration mapping each certificate to each domain.

Instead, Traefik has built in support for *Lets Encrypt*[84]/[38]. This allows the automatic issuance of SSL/TLS certificates for any domains in use by the web server.

3.8 Projects Service - *ProjSvc*

The projects service (referred to as *ProjSvc*) acts as the main backend for any use of the editor UI. *ProjSvc* is responsible managing projects and project deployments.

ProjSvc must be written in TypeScript because it relies on *LangLib* as a dependency which too is written in TypeScript. As a result, *ProjSvc* must use a TypeScript or JavaScript backend engine. Node.js was chosen for this purpose.

3.8.1 *UISvc-ProjSvc* communication

As the frontend editor is served to the user as a *Single Page Application*, there must exist a means for the frontend to communicate with the backend and vice-versa.

In the modern web, a typical approach would be to use *Representational state transfer (REST)* and offer a *RESTful API*[41].

REST generally gives a resource a path and maps *Hypertext Transfer Protocol (HTTP)* verbs onto operations on that resource.

This has some caveats, REST is a commonly implemented *architectural pattern* but it is not *standardised*. While it is built upon standards like HTTP and URI[81] there are only community guidelines[41] discussing how an operation maps to a specific HTTP verb or how a resource is identified by a specific path. As a result while REST APIs are abundant across many products and open-source projects, generally the quality of these APIs can vary from acceptable to quite poor.

This was judged to be of poor design and an alternative standardised solution was investigated for the project.

It was not expected for *ProjSvc*'s API to be consumed externally by other projects or 3rd party services. Therefore, *ProjSvc* would not need to comply with an architectural pattern like REST most demanded by other developers.

Standards such as *GRPC* and *GraphQL* query language both involve writing data definitions in a proprietary *interface definition language* such as *Google Protobuf*[30] or *GraphQL Schema Language*[78]. In addition to this, they involve code generation which complicates build and development steps.

To avoid this, *JSON-RPC*[37] was chosen as the remote procedure call standard. JSON-RPC uses JSON as the serialization standard for carrying procedure call and procedure result from endpoint to endpoint. JSON is supported in most commonly used programming languages and can easily be inlined without code generation or extra build steps.

3.8.2 Database - PostgreSQL

The structured representations used by *LangLib* are expected to be serialized to JSON as they are transmitted from server to client. Therefore, *PostgreSQL* was chosen as the database for *ProjSvc*. because it supports JSON types in schemas natively[65]. Any stored JSON in a column is transformed to a binary format called *jsonb*. Postgres provides extensions to SQL that allow members of the encoded *jsonb* format to be queried as if they were regular members of a schema. This allows for rapid development without the need to continually write new table schemas or write migrations. When development is finalized and the product is ready for release, the JSON keys can be turned into a concrete SQL schema.

3.8.3 Project Management

ProjSvc is responsible for operations that manage projects and their deployments.

When a deployment of a project is requested, *ProjSvc* should create a record of the deployment. A deployment would consist of building a Docker image containing the transpiled Go code of the user project from *LangLib* and any other necessary functionality required. This Docker image would then be ran as a function on *OpenFaaS*.

3.8.4 Traefik Configuration

As mentioned previously in this chapter, Traefik configures itself via the *HTTP provider*.

ProjSvc is responsible for providing the configuration to Traefik. When configuration is requested, *ProjSvc* should query all active deployments from the database. It should provide a configuration that reverse-proxies the domain of each active project to cause the project to execute. When deployments are archived or fail, the entry in the Traefik configuration for that project will be removed.

Chapter 4

Implementation

This chapter discusses the underlying implementation of the design discussed in Chapter 3. Low-level code details such as design patterns, data structures and algorithms are explored. In particular, **technical challenges and how they were overcome is the primary focus of this chapter..** For the scope of the implementation accomplished, see the chapter dedicated to evaluation, Chapter 5.

4.1 Concepts Primer

This section gives a primer on any prevalent implementation patterns. It will be assumed that the reader is familiar with the topics discussed in this section when exploring parts of the implementation later in depth. It will also be assumed the user is familiar with all aspects of the design discussed in Chapter 3

4.1.1 *Head* and *Tail/Mut* structures

The interface definitions discussed in this chapter often are composed of two definitions. A *Head* representing information about that structure such as a identifier, and a *Mut* (sometimes referred to as a *Tail*) that represents the mutable / editable data of that structure. For example, a *Proj* inherits *ProjHead* and *ProjMut*. This is done to separate the record of a resource from the underlying data of the resource. For example, when *ProjSvc* is queried for

a list of projects, it will only return a list of *ProjHead* structures (containing only a project identifier and name for each project) rather than a full *Proj*. A full *Proj* (containing the *ProjMut* with the underlying data of the project) can be returned via a different query. This is done to provide separation of concerns, improve performance and conserve bandwidth.

4.1.2 Wrapper Classes

As the *state* used for project structures needs to be serializable because it is transmitted between frontend and backend, it cannot be an instance of a class in TypeScript. To work around this, *wrapper classes* are used in some areas of the implementation.

A wrapper class constructor consumes a structure/object such as a *Proj* (project representation) and provides a set of utility functions that can be used with that object. These utility functions are generally read only and will not modify the structure. For example, *WProj* provides the *expr(exprId)* function, which is used to look up an expression in the project by its identifier.

The goal of a wrapper class is to reduce code duplication and bloat.

All wrapper classes are prefixed with *W*. For example, the wrapper class for a *Proj* is a *WProj*.

Wrapper classes also provide *static methods* that are used with the *unidirectional data flow* (UDF) pattern (discussed later). These methods are used to modify the object the wrapper class represents.

4.1.3 Identifiers

Identifiers are used on various resources throughout the project. It is often required to keep these identifiers unique when generated both on the backend and frontend, *universally unique* identifiers are used. However, rather than using a traditional *Universally Unique Identifier (UUID)*[66] system, *Universally Unique Lexicographically Sortable Identifiers (ULIDs)* are used instead[86]. This design decision was made because ULIDs can be loosely sorted in the order of the time they were generated. As they have better sorting characteristics than traditional UUIDs they improve query performance when used as indexes (typically *b-trees*) in database tables.

4.2 *LangLib*

As discussed in Chapter 3, *LangLib* is the shared code between *UISvc* and *ProjSvc* for manipulating data structures that represent a project in the visual programming language. *LangLib* contains the important *Transpiler API* which generates Go code from a project.

4.2.1 Type Representation

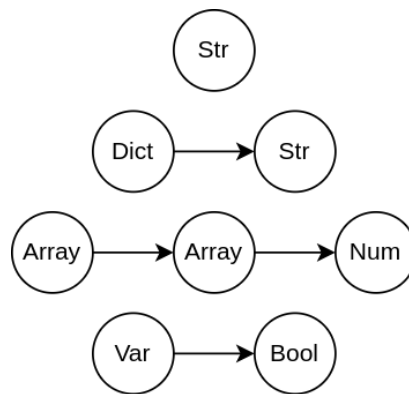


Figure 4.1: Linked-list representations of several types

As discussed in Chapter 3, types can take the form of *container types* or *final types*. As container types contain container types or a final type, a chain structure is required to represent a type.

A *linked list* is naturally an excellent way to model this chained type of structure.

Figure 4.1 shows a visualization of linked lists for several types (*Str*, *Dict*<*Str*>, *Array*<*Array*<*Num*>>, *Var*<*Bool*>>)

A final type does not have a reference to the next link in the chain. If the linked list does not have a final type at its end, the type is not considered *well-formed*. Non *well-formed* types are used within *UISvc* during the process of a user selecting a type with the *Picker* UI element.

Listing 4.1: Type linked list definitions.

```
1 export const enum TypeContainerEnum {
2   Dict = "d",
3   Array = "l",
```

```

4   Var = "&",
5 }
6
7 export const enum TypeFinalEnum {
8   Any = "a",
9   String = "s",
10  Number = "i",
11  Boolean = "b",
12  None = "x",
13  Handle = "h",
14  Model = "m",
15 }
16
17 export interface TypeContainer {
18   container: TypeContainerEnum
19
20   // "?" denotes optional or nullable type in TypeScript
21   contains?: Type
22 }
23
24 export interface TypeFinalBase {
25   final: TypeFinalEnum
26 }
27
28 export interface TypeFinalModel {
29   final: TypeFinalEnum.Model
30   defId?: NamespacedID
31 }
32
33 export interface TypeFinalHandle {
34   final: TypeFinalEnum.Handle
35   defId: NamespacedID
36 }
37
38 export type TypeFinal = TypeFinalBase | TypeFinalModel | TypeFinalHandle
39 export type Type = TypeFinal | TypeContainer

```

Enumerations are used to specify the underlying type category at each link in the list.

In addition to the specification outlined in Chapter 3, two extra final type categories are provided. The *Any* type is used by actions and expressions that use *type inference* for their parameters but have not yet inferred a type. The *None* type is used as an internal type to represent *null* instances of models.

TypeFinalHandle and *TypeFinalModel* are required in addition to *TypeFinalBase* because they need to store a record of the namespace identifier and model/handle identifier (the *NamespacedID*) of the underlying model/handle they represent. The challenge of having many final type structures can be solved by using *union types* in TypeScript. *Union types* allow multiple struc-

tures to represent a single TypeScript type. This can be seen in the listing where *TypeFinal* and *Type* are defined.

Utility functions for handling types are defined on the *WProj* wrapper class. This is required because the logic involving model types needs to be aware of models defined within a project.

Utility functions are available for uses such as type comparisons and constructing types without the need to build linked lists. Examples of some of these functions are shown below.

- `WProj.typeNumber()`
- `WProj.typeBool()`
- `WProj.typeString()`
- `WProj.typeContainer(TypeContainerEnum.Dict, WProj.typeBool())`

Type Comparisons

Implementating type comparisons is not simple and poses a challenge because of *implicit conversions* between language types. The simplest type comparison function implemented is `WProj.typeEqualIff(type1, type2)`. This function is a simple *recursive* walk down the linked lists of each type. At each node in the list, the underlying type category is compared. Additional code ensures handles and model identifiers are equal. Type conversions are not supported in this function.

To support *implicit conversions* and the *Any* type, more complex type comparison functions are built upon the *typeEqualIff* function.

Listing 4.2: Exerpt of type comparison with conversion and *Any* support.

```
1 // Returns true if type 2 is a subtype of type 1 or equal
2 // Supports Any, does not support variable reference conversions
3 typeIsSubType(t1: Type | undefined, t2: Type | undefined): boolean {
4     if (!t1 || !t2) return false
5
6     const t1c: TypeContainer = WProj.typeIsContainer(t1)
7     const t2c: TypeContainer = WProj.typeIsContainer(t2)
8     const t1f: TypeFinal = WProj.typeIsFinal(t1)
9     const t2f: TypeFinal = WProj.typeIsFinal(t2)
10
11     if (WProj.typeIsAny(t1)) {
12         return true
```

```

13     } else if (t1c && t2c && t1c.container == t2c.container) {
14         return this.typeIsSubType(t1c.contains, t2c.contains)
15     } else if (t1f && t2f) {
16         return this.typeEqualIff(t1f, t2f)
17     }
18
19     return false
20 }
21
22 // Returns true if attempt type can be converted into target type
23 // Supports variable reference conversions and Any
24 typeIsCompatible(target: Type, attempt: Type): boolean {
25     if (WProj.typeIsVar(target) && WProj.typeIsVar(attempt)) {
26         return this.typeIsSubType(
27             WProj.typeIsContainer(target)?.contains,
28             WProj.typeIsContainer(attempt)?.contains
29         )
30     } else if (WProj.typeIsVar(target)) {
31         return false
32     } else if (WProj.typeIsVar(attempt)) {
33         return this.typeIsSubType(
34             target,
35             WProj.typeIsContainer(attempt)?.contains
36         )
37     }
38
39     return this.typeIsSubType(target, attempt)
40 }

```

As shown in Listing 4.2 *typeIsSubType* supports the use of comparing against *Any* and can short-circuit the comparison if it appears in the target type.

typeIsCompatible checks for the one *implicit conversion* supported by the language (variable reference to value type, e.g. *Var<Bool>* to *Bool*). If the function detects that a variable reference type is used, it will automatically perform the comparison using the value type of the variable reference if required. It should be noted that a conversion like *Bool* to *Var<Bool>* is **not valid** and should not be mistaken for one. Conversions can only occur in a single direction.

4.2.2 Namespaces

Namespaces have an identifier (e.g. "*number*"), a name (e.g. "*Number*") and a color. A *Namespace* structure stores the records of the actions, expressions, models, handles and triggers within the namespace. *Definition*, *implementation*, and *layout* are stored separately. They are stored in hash tables (known

as type *Record* in TypeScript) indexed by the identifier of the respective element. (e.g. for the *From Str* action, its identifier would be *"from_str"*)

This decision was made to *decouple software components* from the *interface* of the language. In the implementation (*transpiler*), this would allow a move from Go to a completely different target programming language in the future. In *UISvc*, it would allow a move to a completely different user interface framework.

While *LangLib* is shared code, there is a need for namespaces to be extended into *UISvc* as the *layout* functionality relies on the frontend framework (React). There is also a need for a consistent, scalable API to express the standard library. In the future, this API might be consumed by others wishing to add more functionality to the language. To solve this challenge, namespaces are constructed using the *Builder* design pattern[13].

A *NamespaceBuilder* class is provided where builder methods can be called to add elements to the Namespace structure. For example,

exprDef(...), *exprImpl(...)* and *exprLayout(...)* can be called to add definitions, transpiler implementation and UI layout respectively for an expression a user is allowed to use. Similar functionality is available for other language elements such as actions, triggers, models and handles.

4.2.3 Defining Actions, Expressions and Triggers

The *NamespaceBuilder* class is used in *LangLib* to build standard library namespaces (e.g. *HTTP*) with definitions and implementations. The builder is then imported into *UISvc* where layouts are added to the namespace.

Listing 4.3 shows the *NamespaceBuilder* in use defining a small *subset* of the *Number* namespace.

A single action is defined. The *From Str* action which allows converting a string to a number. Two expressions are defined. Expression *Number* allows the use of number literals and expression *+* allows the addition of two numbers. Definitions are accomplished by calls to *actionDef(...)* and *exprDef(...)*

Listing 4.3: *NamespaceBuilder* in use defining a subset of the *Number* namespace

```
1 | export default new NamespaceBuilder(NamespaceID.Number, "Num", "rgba(255,215,0,1)")
2 | .actionDef("from_str", {
3 |     name() { return "From Str" },
4 |     filter: ["From Str"],
```

```

5   params(ctx: ActionCtx): Record<string, Param> {
6       return {
7           "str": {
8               name: "string",
9               type: WProj.typeString()
10          }
11      },
12      branches(ctx: ActionCtx) {
13          return {
14              fallthrough: false,
15              branchClasses: {
16                  "num": {
17                      name: "Number ->",
18                      returns: {
19                          "num": { name: "num", type: WProj.typeNumber() }
20                      }
21                  },
22                  "no_num": {
23                      name: "No Number"
24                  }
25              }
26          }
27      },
28  })
29  .exprDef("num", {
30      name: "Num",
31      filter: ["Number"],
32      options: {
33          "num": { id: "num", type: OptionType.String, default: "0" },
34      },
35      return(): Type {
36          return WProj.typeNumber()
37      }
38  })
39  .exprDef("plus", {
40      name: "+",
41      filter: ["plus", "add"],
42      return(ctx: ExprCtx): Type {
43          return WProj.typeNumber()
44      },
45      params(ctx: ExprCtx) {
46          return {
47              "a": {
48                  name: "a",
49                  type: WProj.typeNumber()
50              },
51              "b": {
52                  name: "b",
53                  type: WProj.typeNumber()
54              }
55          }
56      }
57  }
58  }

```


Definitions have a name and can include additional filter hints (additional search values for fuzzy search).

An action definition has a set of parameters it takes (each with a name and type) and its control flow (*branches*) configuration. The result of an action can *fallthrough* to the next statement or take one or more *BranchClasses*. Each named *BranchClass* has one or more possible named return values.

An expression definition also takes a set of named parameters but only has a single return type.

Action and expression definitions also support *options*. These are a list of properties that can be modified by the user when using a statement/expression. Sometimes these properties are used to store internal values and can be hidden from the user.

Definitions need to be *dynamic*. This means that a definition can react to its use. This is required to support the modification of the definition when options are modified or new information about the definition use is learned.

This challenge is resolved by the use of functions that are called to return values about the definition. These functions can be seen in Listing 4.3 (*params*, *return*, *branches*, *name*). These functions are called with *context* structures (*ActionCtx*, *ExprCtx*) which describe the current statement or expression using the action or expression definition. This context structure can be used to lookup the types of expressions attached to the parameter slots or the values the user has set options to. The definition can then alter itself in response.

Trigger definitions are simply action definitions but take a *TriggerCtx* which contains a trigger-specific structure about the underlying trigger behaviour. A trigger will always appear as the first statement in a function.

Type Inference

Dynamic definitions are used to implement *type inference* and functionality such as dynamic parameters, branches and return types.

For example, in the definition of the *Array* expression (which allows one to construct an array), there must always be an optional free parameter slot at the end of the expression for the user to add more items to the array. In addition to this, the expression must *infer* the array type by examining the types of the expressions the user has placed as elements of the array.

Listing 4.4: Dynamic use of *return* in the *Array* expression

```

1 | return(ctx: ExprCtx): Type {
2 |     return WProj.typeContainer(
3 |         TypeContainerEnum.Array,
4 |         typeArrayConstructorInferContains(ctx)
5 |     )
6 | }

```

Listing 4.4 shows a dynamic *return* implementation for the *Array* expression that infers the array return type. A call is made to *typeArrayConstructorInferContains*. This function uses the *ExprCtx*. An *ExprCtx* provides the expression in use and the *wProj* wrapper class of the project. This means that the call `ctx.wProj.exprParamAttachedTypes(ctx.expr)` can be made to discover the types of attached expressions on the expression parameter slots. These types are then examined and the correct type is returned. For example, if there are no expressions attached, the inferred type is *Array<Any>*. If all attached expression types are *Num*, the inferred type is *Array<Num>*.

Type inference can also be overridden as discussed in Chapter 3. The *typeArrayConstructorInferContains* will check if a *typeSlot* member is set on the *ctx.expr* object and use the overridden type if required.

The approach used for type inference here is deliberately simple to avoid confusing the user. If multiple different types are found, the inference resolves to *Array<Any>*. More complex type inference algorithms can be found in type systems such as the *Hindley-Milner type system*[47] which support *parametric polymorphism*. The type deduction algorithms in these systems can deduce the *most general type* or *principal type* from a hierarchical set of types.

Type inference is primarily used in *LangLib* for deducing types for *Array* and *Dict* expressions and actions which consume those expressions.

4.2.4 Dimensions

A *Dimension* object represents everything in the standard library of the language. A *NamespaceBuilder* can be used alongside the *WDimension* wrapper class to *import* a namespace into the dimension.

A dimension acts a single source of truth and is used to source every aspect of the language the user can consume. In addition to this, stored with a dimension are *FilterMaps* which implement search for all named language elements such as actions and expressions.

Both *ProjSvc* and *UISvc* define and call a function called *langInit()* that sets up the default dimension containing the full language standard library.

4.2.5 Fuzzy Search with *n*-grams

There are many approaches to the challenge of approximate string matching or *fuzzy search*. Approaches can be generalized in the form of *edit distance-based* matching or *token-based* matching. *Edit distance-based* approaches generally use a metric such as *Levenshtein distance* or *Hamming distance*. *Token-based* approaches typically use indexes such as *prefix* or *suffix trees* and score the number of matches. In the implementation, a *token-based* approach was used as it allows for a preprocessed search index to be created to speed up search time. An *edit distance-based* approach would require calculating edit distance for the search string against every name in the search space.

A *n*-gram based character index is used. For example, the character *n*-grams of the string "hello" with $n = 2$ are {"he", "el", "ll", "lo"}. Therefore an index could contain all possible orders of alphabetical strings (and possibly symbols) of length n as its key. The values stored within the index for a particular key reference the strings that contain that *n*-gram.

For example, if there are two items in the corpus and $n = 2$:

1. "To Str" - Expression Definition for *To String* expression
2. "To Bool" - Expression Definition for *To Bool* expression

The index will look like:

```
{
  "to" : [1, 2],
  "os" : [1], "st" : [1], "tr" : [1],
  "ob" : [2], "bo" : [2], "oo" : [2], "ol" : [2]
}
```

A *FilterMap* structure is used to store the value for n and the index. Searching the index is accomplished by splitting the search string into *n*-grams of the index n . The returned search results are ordered by the number of matching occurrences of search *n*-grams against the index. During index creation or index search, if a string is smaller than length n , the string is padded with itself until greater than n . Spaces are stripped.

FilterMaps on *LangLib* provide fuzzy search for the following items on the standard library:

- Action definitions
- Expression definitions
- Handle definitions
- Model definitions
- Model fields
- Types

4.2.6 Project Representation

When engineering a structure to represent a project, a significant implementation challenge was encountered. A traditional implementation of a programming language would likely represent any hierarchical resource in the form of trees. For an example, an expression may be represented by a binary tree object with left and right children.

It was quickly found that such an approach would not work well for the implementation of a visual programming language. This is because any hierarchical element can be referenced directly by the user. For example, a subexpression can be selected, and then dragged and dropped from its parent expression onto the project canvas. This would require a mechanism to identify and reference the subexpression nested deep in the expression tree.

Therefore it was decided that all hierarchical elements would be flattened and referenced directly from the root of the project data structure. Logically, the elements remain as trees but are stored in a flattened form. Any hierarchical element would then simply store a list of its children's identifiers. The structures of the children can then be looked up on the project root structure.

Listing 4.5 shows the definition of a project.

Each element type is stored in a dictionary with the index key as the identifier of that particular element.

This is for hierarchical elements (*blocks*, *expressions*), elements that are contained within elements in a hierarchy (*variables*, *statements*, both found in blocks) and non-hierarchical elements (*functions*, *models*, *model fields*).

Listing 4.5: The data structure representing a project.

```

1 export interface ProjMut {
2   namespaceId: string
3
4   funcs: Record<string, Func>
5
6   blocks: Record<string, Block>
7   statements: Record<string, Statement>
8   variables: Record<string, Variable>
9   exprs: Record<string, Expr>
10  models: Record<string, ModelDef>
11
12  fieldNoNext: number
13
14  fields: Record<string, ModelField>
15
16  jumpsFrom: Record<string, Record<string, Jump>>
17  jumpsTo: Record<string, Record<string, Jump>>
18  environments: ProjEnvs
19
20  version: number
21 }

```

To ease the challenge of working with the flattened structure, several helper methods are available. The *WExpr* wrapper class provides a method called *WExpr.exprTreeVisit(exprs, rootExprId, visitor)*. This uses the *Visitor* design pattern[13] to allow a depth-first recursive visit of every expression in a flattened expression tree from a specified root expression. The visitor function can be passed which is called for each *Expr* in the tree.

WProj.blockScope(block) will start from a block and follow its parent. It will traverse up the block tree until it reaches the root block of a function. At each block, it will record the variable identifiers in that block and add them to a dictionary. This dictionary will be returned and can be used as a lookup table to check and enforce variable scopes.

Blocks and Variables

A *Block* structure stores its block identifier, its parent block identifier, and lists of identifiers of the child blocks, variables and statements in the block

A *Variable* structure stores the variable identifier, name and type.

Statements and Expressions

Statement/Expr structures store a *defId* (*NamespacedID*) pointing to the namespace and identifier of the action/expression definition they use.

For representation of children, statement structures have a dictionary called *paramSlots*. This dictionary is indexed by the same key as the dictionary returned by a call to *params()* of the action definition the statement uses. The value stored for a key is the identifier of the expression in that particular slot. If the key is absent, the slot is empty.

This same functionality exists for expressions and their parameter slots.

Both *Statement* and *Expr* structures have an *options* field which contains the values of the options defined by the action/expression definition they rely on.

As *Exprs* can *float*, they have a *pos* attribute storing their position on the graph. This field is *optional* as an expression should not be able to float when placed in a slot.

Control Flow

Representing control flow was a challenge because it is a mix of two concepts. When a statement is executed it can either *fallthrough* (if allowed by its action definition) or take one of many possible branches.

When a *fallthrough* occurs, execution will normally pass to the next statement in the current block.

Branch control flow is implemented via a *directed graph* stored in the *adjacency list* form. Dictionaries are used rather than lists for $O(1)$ amortized lookup time when checking for the presence of branches (sometimes referred to as an *adjacency map*). Structures like *adjacency matrices* were avoided because the branch graph is generally sparse and this would waste memory. The two adjacency maps can be seen stored on the project structure in Listing 4.5 as *jumpsFrom* and *jumpsTo*.

jumpsFrom is a nested map, first indexed by the origin statement identifier and then indexed by the origin branch identifier.

jumpsTo is similar, first indexed by the target statement identifier and then indexed by the jump identifier.

Listing 4.6 shows the Jump data structure which represents a graph edge.

The branch identifier is the identifier of the branch returned by *branches()* on the action definition of the origin statement.

Listing 4.6: Jump structure.

```
1 export interface Jump {  
2     jumpId: string  
3     origin: {  
4         statementId: string  
5         branchId: string  
6     }  
7     target: {  
8         statementId: string  
9     }  
10 }
```

The target statement for a jump will use the action definition *jumpTarget*. The *jumpTarget* action definition can dynamically read information about the jump and return parameters in its *params* function that match the parameters returned by the *BranchClass* defining the branch being taken.

Similar to the tree structures in a project, helper methods are also provided when working with the *control-flow graph*.

`WProj.reachableVisit(rootStatement, visitor)` is a helper method that will traverse the entire control flow of the project depth-first. It will respect *fallthrough* semantics and look up the next statement in the current block if required. It will visit every reachable statement from the specified root statement. Statements will only be visited once to prevent cycles and infinite loops. This helper method is used when validating that all statements in the project are type-checked and for performing *dead-code analysis*.

Project Validation

Validating a project is easily accomplished with the visitor methods discussed above.

Every single reachable statement in the project is visited. The action definition of each statement is looked up and the type of each expression in each parameter slot is type checked (using `WProj.typeIsCompatible(t1,t2)`) against the types required by the action. Variable expressions are checked to ensure they remain inside the scope they are entitled to.

A branch is not required to have a jump so this does not need to be checked.

A list of problems with lookup tables referencing the error location is built. Problems can include invalid types, missing required parameters or variables being used outside of their scope.

If no problems are found, the project is considered valid.

ModelDefs and ModelFields

A *ModelDef* structure represents a user defined model. It has a member dictionary called *fields* that contains the identifiers of fields within the model.

A *ModelField* structure defines a named field with a type. The field can have *extensions* which is a structure that describes the user-specified JSON serialization key.

There is a need for model fields to be serializable in all cases and remain forwards compatible. This is because instances of models may need to be transmitted to an internal service, e.g. when passing model instances to Node.js code from the language.

This challenge is solved by assigning every field a *unique constant field number*. No two field numbers should ever conflict within a project. This allows serialized models to be transmitted internally during execution without relying on the JSON serialization keys defined by the user. This approach is used by *Google Protobuf*[30] to allow fields to be renamed without interfering with system operation as field numbers never change.

The next field number (*fieldNoNext*) that can be used is always stored at the project root. When a field is added, *fieldNoNext* is incremented. One downside of this approach is that this places an arbitrary limit on the number of fields available in a project at $2^{53} - 1$.

As mentioned in Chapter 3, models and fields can be archived. This is accomplished with a simple boolean flag.

ProjEnvs

A *ProjEnvs* structure contains a list of dependencies for each traditional programming language environment supported by the language. Currently it only stores the same format as the *dependencies* key of the Node Package Manager (npm) *package.json* format. These dependencies are installed during the deployment stage of a project. (discussed later in *ProjSvc*)

Func

A *Func* structure only contains identifiers of the function, its root block and trigger statement.

4.2.7 Project Mutations with Unidirectional Data Flow (UDF)

Enforcing valid mutations and referential integrity on data structures is **very difficult**. The *unidirectional data flow* or *state, action, reducer*[67] architectural pattern is used to overcome this challenge. It allows for strictly defined mutations to be applied to a data structure. This pattern was first popularized by Facebook with *Flux*[19] for building scalable applications.

There are 3 core concepts to this pattern:

1. *State* - State is stored in an immutable, serializable data structure
2. *Actions* - Actions are objects that describe an event that occurred in the context of the current state
3. *Reducer* - Reducers are functions that consume an action and output a mutated copy of the previous state implementing the required changes requested by the action

In the example of *LangLib*, it provides the *state*, e.g. a data structure representing a project. To create a new statement in the project, an *action* such as *StatementCreate* is instantiated with various parameters, e.g. what block the statement should be placed in. This action is then passed to the *reducer* function, which will *consume* the action and output a copy of the *state* with the new statement added.

Reducers are *pure* and *idempotent*[68]. This means that a reducer has no side-effects and will produce the same output for a given input. An action can result in a large amount of mutations happening to the state but if reducer purity is enforced, then it is guaranteed that the state is always known-good and valid. The pattern could be viewed very similarly to an *atomic transaction* in a database.

Listing 4.7 shows a list of all possible project actions. These structures define the parameters of that specific action. Many are self-explanatory, some

will be explained where they are used in other parts of the implementation (such as in *UISvc*).

Listing 4.7: All possible *ProjActions*

```

1 export type ProjAction =
2   ProjActionEnvs |
3   ProjActionFuncCreate |
4   ProjActionFuncDelete |
5   ProjActionFuncCollapse |
6   ProjActionFuncTrigger |
7   ProjActionFuncTriggerOptions |
8   ProjActionStatementAdd |
9   ProjActionStatementDelete |
10  ProjActionModelCreate |
11  ProjActionModelArchive |
12  ProjActionModelCollapse |
13  ProjActionModelFieldAdd |
14  ProjActionModelFieldExtensions |
15  ProjActionModelFieldArchive |
16  ProjActionFieldNoNextSet |
17  ProjActionBlockAdd |
18  ProjActionBlockDelete |
19  ProjActionVariableAdd |
20  ProjActionVariableDelete |
21  ProjActionJumpAdd |
22  ProjActionJumpDelete |
23  ProjActionExprEnvelopeAttach |
24  ProjActionExprEnvelopeDelete |
25  ProjActionExprDelete |
26  ProjActionExprOptions |
27  ProjActionExprTypeSlot |
28  ProjActionStatementOptions |
29  ProjActionStatementEnvelopeAttach |
30  ProjActionStatementEnvelopeDelete |
31  BatchAction<ProjAction>

```

A *BatchAction* allows for a group of *ProjActions* to be combined and mutate the project structure as one atomic transaction. This is used when adding a function to a project. A function requires a root block and a trigger statement to exist. A batch action is therefore required which would be composed of a *BlockAdd*, *StatementAdd* and *FuncCreate* action. To reduce complexity, helper functions known as *action creators*[69] can be used to create an action or batch action. `WProj.ac_funcCreate(funcId, name)` is the helper function that would be used in this case.

Listing 4.8: *WProj* reducer

```

1 static reduce(p: Proj, a: ProjAction) {
2   const action = a.action
3   const s = makeKey("a_", action)

```

```

4 |     WProj[s](p, a as any)
5 |     p.version += 1
6 | }
7 |
8 | static a_variableAdd(p: Proj, a: ProjActionVariableAdd) {
9 |     if (!(a.variable.blockId in p.blocks)) return
10 |
11 |     p.blocks[a.variable.blockId].variableIds.splice(
12 |         a.index, 0, a.variable.variableId
13 |     )
14 |
15 |     p.variables[a.variable.variableId] = a.variable
16 | }

```

Listing 4.8 shows a *reducer* function (on *WProj*) that extracts the action type and then delegates the call to another function. If *action* is *variableAdd* then *a_variableAdd* will be called. As seen, adding a variable modifies two different data structures. As these modifications are a single mutation in the context of the action, referential integrity is enforced and the project structure will always remain valid.

This enforcement of referential integrity is used in many places. One major example is removing invalid jumps when a statement referenced by the jump is removed.

There is a need for cached data (computationally heavy) within *UISvc* and *ProjSvc* to become aware when it is stale. This challenge is solved by the addition of a *version* attribute to the *Proj* structure as shown in Listing 4.5. The version number is incremented every single time the project is mutated in the project reducer. When the version number is incremented it signals to those who consume the structure that any cached data they have must be recomputed.

4.2.8 Transpiler

The *Transpiler* generates Go code from a project. All transpiler functionality is located on a class called *EmitterCtx*. A new *EmitterCtx* instance must be constructed for each project transpilation.

Code generation is a significant implementation challenge. As discussed in Chapter 3, control flow always takes an explicit path specified by the user.

Mapping this behaviour to a traditional programming language like Go is quite difficult. This is because control flow can take sudden jumps within a block or jump to a parent or child block explicitly (which is almost always

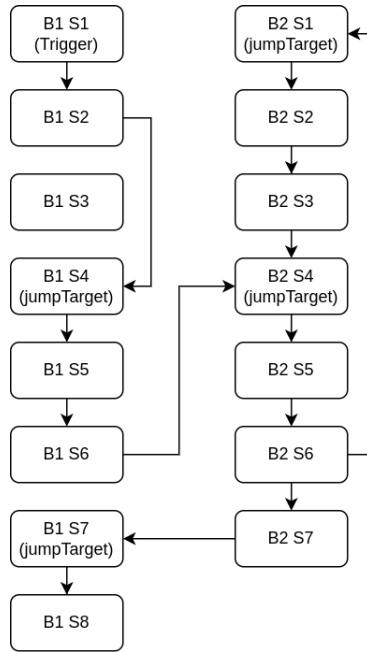


Figure 4.2: Control-flow graph of a sample function

done implicitly in a traditional programming language). Figure 4.2 shows an example of a control flow graph representing the possible execution paths of a function. This graph has two blocks each with their own set of statements. B_i represents block i , S_j represents statement j . For example, $B1 S1$ is statement 1 in block 1. This statement is the trigger statement and is the entrypoint for execution.

The transpiler generates a *finite state machine* in Go from the control-flow graph. A state contains Go code that is the transpilation of a contiguous set of statements and expressions. This contiguous set of code is known as a *sequence*.

To build the set of *sequences* required to generate the state machine, the control-flow graph is divided into *disjoint sets of statements*. Therefore each statement in the control flow graph can only be a member of one sequence.

The sequence generation algorithm traverses the control-flow graph. Each *jumpTarget* statement (a statement where a branch can jump to and begin executing), marks the entrypoint for a new state. Therefore the current sequence ends and a new sequence begins. Each following fallthrough statement

is added to the sequence. Branches to other statements are ignored. This means that every sequence will only contain statements from the same block. Any unreachable statement is excluded.

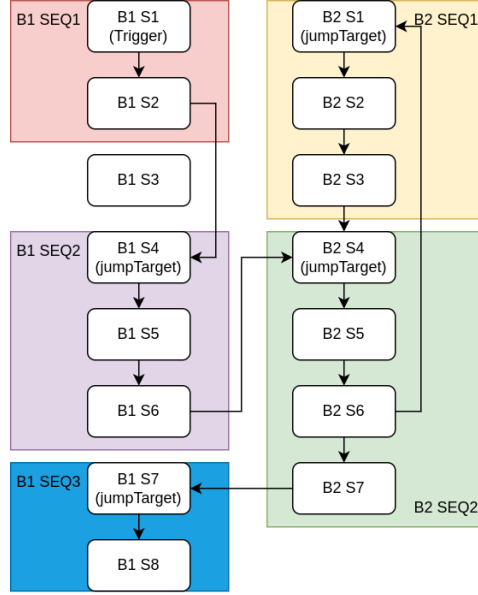


Figure 4.3: Control-flow-graph grouped into *sequences*

Figure 4.3 shows a representation of the control-flow graph shown in Figure 4.2 after each statement has been placed into a sequence. Each sequence is identified by B_i where i is the block number and SEQ_j is the number of that sequence in the block (the *sequence number*).

The state machine stores the current state as the *block number* and a *sequence number*.

During code generation, the block tree of the function is walked and at each block a switch statement is generated with cases for each *block number* of the blocks children.

Within each case is another switch statement with cases for each *sequence number* within that block. Each case has the transpilation of the statements and expressions within that sequence inserted.

Labels (for *gotos*) are placed at the entry into each switch statement. When one wants to jump to another *sequence*, the block number is set to the target block (the currently executing block, the parent block or a child block).

The sequence number is set to the target sequence. A `goto` statement is then inserted that jumps to the switch statement for the target parent, current or child block. This switch statement is evaluated and execution moves to the correct sequence. This implements the full state machine functionality.

Branch return values are implemented by a *jumped* variable that stores a structure containing return values for the last jump.

Type Mappings

Standard types like *Str*, *Bool*, *Num* are mapped directly to their Go equivalents (*string*, *bool*, *float64*) *Array* and *Dict* types are mapped to Go arrays and maps.

Model definitions are converted to Go structure definitions. All model and handle instance variables are stored as pointers to an instance of underlying model/handle structure. Go uses garbage-collected pointers so as model/handle instance variables are assigned, unreferenced structures are automatically freed.

Variable references (e.g. *Var<Bool>*) are simply pointers. For example: *Var<Bool>* is transpiled to type `*bool` and *Var<SomeModel>* is transpiled to `**m_modelId` where *modelId* is the model identifier.

Variables

Variables definitions are inserted and assigned default values at the entry to block switch statements in the state machine. This means that the exact same scope semantics enforced in the language are also enforced in the target transpilation language.

4.2.9 Implementing Actions, Expressions and Triggers

Each language function is converted into a Go function (containing the generated state machine). The parameters of this Go function are returned by a call to the *params()* function on the *triggerImpl* of the trigger used by the function.

For example, the *http::On Request* trigger returns `w http.ResponseWriter, r *http.Request` as parameters which are the same parameters used by

HTTP request handlers in Go. This clearly implies that this language function will be used to handle a HTTP request.

A *setup()* function on the *triggerImpl* is called and is passed the *EmitterCtx*. This allows the trigger to add Go code which calls the Go function generated earlier. In the case of the *http::On Request* trigger, it registers the Go function in a route-handler for the request method and path defined by the options on the trigger.

A *body()* function on the *triggerImpl* is then called which is responsible for emitting the implementation body of the trigger statement. The body of the *http::On Request* trigger does sanitation of the request and sets passed variable references to values from the request as configured by the user.

Listing 4.9 shows the implementation of the *From Str* action and the *Plus* expression for the *Num* type. The *EmitterCtx* has a range of methods. The most important function is *line* which can be used to add a line to the current transpilation output. Calls to *statementParam* and *exprParam* can be used to recursively transpile expressions that are passed as parameters in parameter slots. These expressions can then be inserted in a line of code as seen in the listing. Imports of Go modules can be done the *m* function and the creation of temporary variables with *tempVar* methods. Action branches can be taken by passing a valid jump structure to *jump*. *def* can be used to add top-level definitions to the program (e.g. structures). Like definitions, it should be noted that *emitter* is a dynamic function, transpilation output can change depending on the options configuration and types of parameters in the action or expression.

Listing 4.9: *From Str* action and *Plus* expression implementation

```

1 | .actionImpl("from_str", {
2 |     emitter(em: EmitterCtx, ctx: ActionCtx) {
3 |         const num = em.wProj.jumpFrom(ctx.statement, "num")
4 |         const no_num = em.wProj.jumpFrom(ctx.statement, "no_num")
5 |
6 |         const strc = em.m("strconv")
7 |
8 |         const t = em.tempVarRaw("float64")
9 |         em.line(
10 |             '${t}, err = ${strc}.ParseFloat(${em.statementParam(ctx, "str")}, 64) '
11 |         )
12 |         em.line('if err != nil {'
13 |         if (no_num) {
14 |             em.jump(no_num)
15 |         }
16 |         em.line('} else {'

```

```

17         if (num) {
18             em.jump(num, { "num": `${t}` })
19         }
20         em.line('}')
21     }
22 })
23 .exprImpl("plus", {
24     emitter(em: EmitterCtx, exp: ExprCtx) {
25         return `${em.exprParam(exp, "a")} + ${em.exprParam(exp, "b")}`
26     }
27 })

```

The implementation of *From Str* adds a call to `strconv.ParseFloat` to convert the passed *Str* to *Num*. Different jump paths are taken on whether the parsing worked or failed.

4.2.10 Programming Language Interoperability

Interoperability with other programming language environments is achieved by a *process manager*.

When the transpiled Go server process boots, it will then boot up any needed programming language environments. These processes are managed simultaneously alongside any executions of function state machines. If an environment crashes, it is automatically rebooted.

This is accomplished by using a *goroutine* sitting on a different thread of execution watching the process. This reinforces the rationale for choosing Go as outlined in Chapter 3. Currently Node.js is supported as an interoperable traditional programming language. The Node.js process is booted and runs a script that contains a simple HTTP handler that can receive JavaScript payloads, execute them with `eval` and return the result.

The *Node.js Code* action definition implements the *variable capture list* behaviour as discussed in Chapter 3. The action implementation generates a JavaScript header that contains serialized forms of the variable values in the capture list. This header is concatenated onto the JavaScript the user wants to run and is transmitted to the *Node.js eval* server discussed above. The code is ran by the server and a response payload is returned.

The returned response payload is deserialized and captured variables are updated to their new values.

4.3 *ProjSvc*

As discussed in Chapter 3, *ProjSvc* is the main backend that is interacted with when the user is using *UISvc*. *ProjSvc* is responsible for managing projects and project deployments.

4.3.1 JSON-RPC

As mentioned in Chapter 3, *JSON-RPC* was chosen as the remote procedure call protocol between *ProjSvc* and *UISvc*.

A typical RPC request and response is shown in Listing 4.10 and Listing 4.11:

"*jsonrpc*" specifies the protocol version.

"*method*" specifies the serverside method to call.

"*params*" specifies a JSON object to be passed as parameters to the serverside method

"*id*" is the request identifier.

"*result*" is the result JSON object.

Listing 4.10: JSON-RPC Request Payload.

```
1 {  
2     "jsonrpc": "2.0",  
3     "method": "subtract",  
4     "params": {  
5         "a": 10,  
6         "b": 5  
7     },  
8     "id": 1  
9 }
```

Listing 4.11: JSON-RPC Response Payload.

```
1 {  
2     "jsonrpc": "2.0",  
3     "result": 5,  
4     "id": 1  
5 }
```

The specification also provides support for error handling with error types allocated to *implemented-defined* errors and *application-defined* errors.

Interface definitions that define the contents of the "*method*", "*params*" and "*result*" field are shared between *ProjSvc* and *UISvc*. For simplicity,

LangLib is used for this as it is already used for sharing code between these two codebases.

Strictly coupling to HTTP as a transport protocol should be avoided. To overcome this challenge, a custom JSON-RPC library was written rather than using an external library. This was done to *decouple transport* of the payload from *protocol*. An external library that does this could not be found. The current architecture uses a client to server model where *UISvc* only makes remote procedure calls on *ProjSvc* (via the *JSONRPCTransportHTTP* transport). Transport is *pluggable* for future *forward compatibility*. For example, a *JSONRPCTransportWebSocket* transport could be introduced that allows *ProjSvc* to make remote procedure calls on a client *UISvc* connected over a *WebSocket*[54].

An *RPC* class is passed an interface type as its generic parameter. This interface has the method and parameter definitions of the service. *RPC* provides methods to deserialize, parse, sanity check and serialize payloads in the interface when received on the supplied transport.

Listing 4.12 shows the full list of methods that *UISvc* or any other service consuming *ProjSvc* can call over *JSON-RPC*

Listing 4.12: *ProjSvc* available RPCs.

```
1 export interface ProjService {
2   projRead(p: APIProjReadParams): APIResult<ProjHead[]>
3
4   projCreate(p: APIProjCreateParams): APIResult<APIOK>
5   projRead(p: APIProjReadParams): APIResult<Proj>
6   projUpdate(p: APIProjUpdateParams): APIResult<APIOK>
7   projDelete(p: APIProjDeleteParams): APIResult<APIOK>
8
9   deploymentsRead(
10     p: APIDeploymentsReadParams
11   ): APIResult<ProjDeploymentHead[]>
12   deploymentCreate(p: APIDeploymentCreateParams): APIResult<APIOK>
13   deploymentRead(p: APIDeploymentReadParams): APIResult<ProjDeployment>
14 }
```

4.3.2 Projects and Deployments

Project RPCs (e.g. *projRead*, *projCreate*) are *CRUD* (*create*, *read*, *update*, *delete*) operations that involve only simple SQL queries to the database.

Deployment RPCs are more complex and involve long running tasks and jobs (0.5 seconds to 5 minutes) When a deployment of a project is

created (via *deploymentCreate*), a deployment record is inserted into the database. A deployment record has a copy of the project being deployed, the current deployment status and a deployment log tracking the progress of the deployment.

Creating a deployment generally returns a result immediately to the caller. This is because the deployment is initially created with the *Waiting for deployment* status.

An *asynchronous* task on *ProjSvc* polls the database for deployments with the *waiting for deployment* status. When it finds a deployment waiting, it will start the deployment. Any calls to *deploymentRead* will return the current deployment status so a user can watch the deployment as it progresses.

The set of stages taken by *ProjSvc* during a deployment are the following:

1. Mark any *active* deployments in the database as *archived* and no longer in use.
2. Mark the new deployment status in the database as *deploying*.
3. Call into *LangLib* and transpile the target project into Go code.
4. Create the contents of the container build directory, including the transpiled Go source code and image build *Dockerfile*.
5. Send the contents of the directory to the *Docker daemon* and request that an image is built using the *Dockerfile* and build directory contents.
6. Push the built image to the image registry.
7. Create an function in OpenFaaS that uses the built image from the image registry.
8. Wait for the project to return the healthy status when queried via HTTP.
9. Mark the deployment as *active*.

This allows the project to be added to Traefik/reverse-proxy configuration (discussed later).

An *AbortController*[48] is used to prevent any of the deployment stages stalling indefinitely. A deployment will automatically time out and receive the *timed out* status if the deployment has not finished within 5 minutes.

At each deployment stage, the log is saved to the database. If the deployment fails at any time the deployment status is marked as *failed*.

4.3.3 Improving Deployment Size and Time

A significant implementation challenge was reducing the deployment time of a user project. The easiest avenue for this was reducing the Docker image build time and size. By reducing the image size, the container can be uploaded and downloaded faster from the registry. This reduces the amount of time *OpenFaaS* spends setting up the container.

Listing 4.13 shows the *Dockerfile* used for building a user project into an image.

Listing 4.13: Project build *Dockerfile*

```
1 FROM golang:1.17-alpine3.14 as build
2
3 COPY ./go.mod ./go.sum /tmp/deployment/
4 RUN cd /tmp/deployment && GO111MODULE=on GOPROXY="" GOFLAGS="" DEBUG=0 go mod download
5
6 COPY . /tmp/deployment/
7 RUN GO111MODULE=on GOPROXY="" GOFLAGS="" DEBUG=0 cd /tmp/deployment && \
8     go build --ldflags "-s -w" -o ./projbin main.go
9
10 # of-watchdog is the base Go image needed to be used with OpenFaaS
11 FROM ghcr.io/openfaas/of-watchdog:0.9.2 as watchdog
12 FROM alpine:3.14
13 RUN apk --no-cache add ca-certificates libc6-compat nodejs npm \
14 && addgroup -S app && adduser -S -g app app
15 USER app
16
17 COPY --chown=app:app --from=build /tmp/deployment/node /tmp/node
18 RUN cd /tmp/node && npm install
19
20 COPY --chown=app:app --from=watchdog /fwatchdog /tmp/fwatchdog
21 COPY --chown=app:app --from=build /tmp/deployment /tmp/
22
23 ENV fprocess="/tmp/projbin" mode="http" upstream_url="http://127.0.0.1:8082" prefix_logs="false"
24 CMD ["/tmp/fwatchdog"]
```

Multi-stage builds

The first approach to reducing image size was to make use of *multi-stage builds*[10]. Multi-stage builds allow one to perform a build stage in one particular image, then use an alternative base image but copy files from the previous build stage. This is seen in the listing where an image with the full Go compiler (`FROM golang:1.17-alpine3.14 as build`) is used to build the transpiled Go code. The final base image used is *alpine:3.14* on the final `FROM` statement. The `/tmp/deployment` folder is copied from the stage named `build` to the new base image. This means the final base image has no Go compiler or related utilities installed and as a result is much smaller.

Minimal container images

Alpine Linux base images are used rather than other distributions. Alpine is deliberately built to provide a small footprint. It uses *musl*, a minimal C runtime rather than *glibc* (GNU Lib-C, the standard C library and runtime found in most Linux distributions). For comparison, an *alpine:3.11* image is only 2.7MB compared to a 63MB *fedora:31* image[4]

Build layer caching

Docker runs each command in intermediate containers and performs *build caching*[88]. Each command in the Dockerfile is considered a new layer. Each layer is then stored in a local cache by the Docker daemon. When an image build is ran, cached layers are used automatically. When an `ADD` or `COPY` instruction is found, the files specified are hashed. If the files are found to be unchanged, the cached layer is used instead.

The layer cache applies from the current layer to the base image of the current build stage. With this knowledge we can establish what is always cached in our build:

- Go module and sums file
- Go module dependency downloads (the dependencies and versions are always fixed)
- Installation of `ca-certificates` `libc6-compat` `nodejs` `npm` packages

- Adding `app` user and group

The following layers are conditionally cached:

- Go build of transpiled code

This layer will be cached if the user performs multiple deployments of a project that has not been modified.

- Node.js dependencies

This layer's cache is invalidated if the user modifies the Node.js environment dependencies on their project (rare)

Go linker tuning

When using the Go compiler during a build, two additional flags are passed to the Go linker. `-w` and `-s` build the binary without the DWARF and Go specific symbol tables. These occupy a lot of space and are not needed because it is not expected for these built binaries to ever be ran inside a debugger.

Tmpfs filesystem builds

When *ProjSvc* is writing the deployment files to be uploaded to the build daemon alongside the Dockerfile, it does this in a *tmpfs* filesystem (*/tmp*).

The Linux kernel stores *tmpfs* filesystems in memory, swapping out pages that are least-used to disk. By building the deployment files in a *tmpfs*, the process is limited by memory speed rather than the speed of any I/O storage devices.

4.3.4 Traefik Configuration

Traefik is configured to poll the `/lb-config` path over HTTP on *ProjSvc*. It expects to receive a JSON response with the configuration it will use.

Execution of a function via the OpenFaaS API is done via HTTP requests to the `/function/functionName/` endpoint on *faasd* where *functionName* is the name of the function. All requests must be sent to *faasd* which then passes the request to the container. Anything after the path is passed as the base path to the HTTP handler inside the container.

This introduces the challenge that URL rewriting will be required to reverse-proxy a request from a project subdomain to the OpenFaaS daemon. Traefik is configured via *routers*, *middleware* and *services*. Routers are entry-points that take in requests, can optionally pass them through middleware and then send them to a service.

To solve the problem, the *AddPrefix* middleware is used which can add */function/functionName/* prefix to the request path and then send it to *faasd*

An example of the JSON returned by *ProjSvc* for the reverse-proxying of active deployments is shown in Listing 4.14. A single project, *testmodel* is deployed.

Listing 4.14: Autogenerated Traefik configuration

```

1 {
2   "http":{
3     "routers":{
4       "webpl_router_testmodel":{
5         "entrypoints":"websecure",
6         "rule":"Host('testmodel.webpl.test')",
7         "service":"webpl_service_testmodel",
8         "tls":true,
9         "middlewares":[
10          "webpl_middleware_prefix_testmodel"
11        ]
12      }
13    },
14    "middlewares":{
15      "webpl_middleware_prefix_testmodel":{
16        "addPrefix":{
17          "prefix":"/function/testmodel"
18        }
19      },
20    },
21    "services":{
22      "webpl_service_testmodel":{
23        "loadBalancer":{
24          "servers":[
25            {
26              "url":"http://faasd.test:8080/"
27            }
28          ]
29        }
30      }
31    }
32  }
33 }
```

4.4 *UISvc*

UISvc is the Single Page Application served to the user that allows them to edit and deploy a project in the visual programming language. This section is concerned about *UI implementation*, not *UI appearance and design*. Descriptions and figures of visual elements in *UISvc* can be found described in Chapter 3.

4.4.1 Component Design

A UI component library (which offers pre-made components like *Buttons*, *Sidebars*, *Menus*) was not used. All user-interface elements are original, from scratch implementations. This involves more work but is more aesthetically pleasing and allows for more control over the final visual design.

A significant downside to this approach is that CSS styles need to be placed into dictionaries and passed to the *style* prop of each component. The workaround is the use of the `styled-components`[87] library. It allows one to mix inline CSS and prop values to quickly create components. Listing 4.15 shows `styled-components` in use. *UIFloating* is an element that can be positioned at any arbitrary *x* and *y* coordinate relative to its parent element. `styled-components` allows the position to be inlined directly into CSS rather than having to generate a style dictionary.

Listing 4.15: *UIFloating* implementation

```
1 | const UIFloating = styled.div`  
2 |   position: absolute;  
3 |   top: 0;  
4 |   left: 0;  
5 |  
6 |   transform: ${({p: UIFloatingProps}) => `translate(${p.pos.x}px, ${p.pos.y}px)`};  
7 | `
```

4.4.2 Editor State

Editor state is managed using the *unidirectional data flow* pattern as was used for managing project state in *LangLib*. It is used for the same reason: enforcing valid mutations and referential integrity.

Listing 4.16 shows the *Editor* and *EditProj* structures, these define the global state of the editor. They act as a single source of truth for current

editor layout and behaviour. For example, *editProj* is an optional field on *Editor*, this is because *editProj* is not present when a project is not being edited.

A list of all editor actions can be found in the listing. Of importance is *EditorActionEditProjAction*, this action contains another action of type *EditProjAction* and will perform the action on and mutate the current *editProj* structure on *Editor*.

The *proj* field on *EditProj* is the *LangLib* project currently being edited. This *proj* field is mutated by *ProjActions* (as was discussed in *LangLib*). This is achieved by using the *EditProjActionProjAction* which has a field for the *ProjAction* which will be used to mutate the project.

Listing 4.16: Editor state structures

```

1 export interface Editor {
2   projs: Record<string, Proj>
3   editProj?: EditProj
4   errors: EditorError[]
5 }
6
7 export interface EditProj {
8   proj: Proj
9
10  addJumpMode?: AddJumpMode
11  selected: Selected
12
13  graphPan: Vec2
14  graphZoom: number
15
16  validate?: Validate
17  visible?: Trace
18
19  typesFilter?: FilterMap<Type>
20  variablesFilter?: FilterMap<string>
21  modelsFilter?: FilterMap<string>
22  modelFieldsFilter?: FilterMap<string>
23
24  picker?: Picker
25  dragType?: Type
26 }
27
28 export type EditorAction =
29   EditorActionProjLoad |
30   EditorActionProjsLoad |
31   EditorActionEditProjAction |
32   EditorActionErrorShow |
33   EditorActionErrorDismiss
34
35 export type EditProjAction =
36   EditProjActionProjAction |

```

```

37 | EditProjActionExprMode |
38 | EditProjActionSelected |
39 | EditProjActionGraphPan |
40 | EditProjActionGraphZoom |
41 | EditProjActionAddJumpStart |
42 | EditProjActionAddJumpEnd |
43 | EditProjActionAddJumpHover |
44 | EditProjActionVisibleRebuild |
45 | EditProjActionPicker |
46 | EditProjActionDragType |
47 | EditProjActionValidate |
48 | EditProjActionTypesFilterRebuild |
49 | EditProjActionVariablesFilterRebuild |
50 | EditProjActionModelsFilterRebuild |
51 | EditProjActionModelFieldsFilterRebuild

```

The *EditProj* structure is responsible for keeping track of the following aspects:

- Current project (the *LangLib* structure representing the project)
- Current project validation (list of errors and their locations)
- Visibility trace (reachable statements in the control flow graph)
- Current graph pan and zoom (of the zoomable user interface)
- Picker configuration (what Picker layout needs to be visible, if any)
- Jump addition (adding jumps is a 2 step process, click on origin, then target)
- *dragType*, the type of the currently dragged expression (if any)
- FilterMaps for aspects related to the current project

The use of *FilterMaps* in *LangLib* provides fuzzy search for all aspects of the language standard library. The *FilterMaps* on the *EditProj* structure provide fuzzy search for elements in the current project. Fuzzy search can then be provided for user-defined types, variables, models and model fields that are not found in the standard library.

The global *Editor* structure is stored at the *UIEditor* component. It is stateful (i.e it is managed by a *useState* hook, as discussed in subsection 3.1.1). *editorAction*, *editProjAction*, and *projAction* functions are defined. These

functions mutate the *Editor*, *EditProj* and *Proj* structures by accepting an action, mutating the structure and calling *setState* to update the editor state.

When the editor state is updated, this triggers a rerender which causes the React tree-diffing algorithm to determine which components in the application need to be rerendered with new properties.

4.4.3 React Contexts

There is a need for the global editor state to be available and injected into every UI component that requires it. One way of achieving this is to pass the editor as a *prop* into every component and its children. This is not flexible and requires large complex code modifications. To resolve this challenge, React has *contexts*. A context allows one to provide an object from a higher part of the DOM/component tree for it to be consumed by one of its deeply nested children.

For example, one of the contexts written for *UISvc* is the *ProjServiceContext*. The *ProjServiceRPC* class (which allows for communication with *ProjSvc*) is stored in the root component of the application (*UIEditor*). All other components in the editor are children within the *UIEditor* component. The *ProjServiceContext* can be *provided* the instance of the *ProjServiceRPC* class within the root *Editor* component.

This means any child component (even deeply nested) within the *UIEditor* component tree can simply use

`const projSvc = useContext(ProjServiceContext)` to gain access to the *ProjServiceRPC* object.

For example, if they then wanted to update the project they can simply use `projSvc.rpc("projUpdate", proj)`

Several contexts are used in *UISvc*:

- *EditorContext*

Provides the *Editor* structure, a *WEditor* wrapper class and a *editorAction* function for running editor actions to mutate editor state.

- *ProjServiceContext*

Provides the *ProjServiceRPC* object for making calls to the backend

- *GraphContext*

Provides the scale and current mouse position on the pannable and zoomable canvas.

- *ProjContext*

Provides the *EditProj* and *LangLib*'s *Proj* structures, the *WProj* and *WEditProj* wrapper classes. A *projAction* function is provided for mutating the current project according to the actions described in the *LangLib* implementation. *editProjAction* is provided for mutating the *EditProj* structure

- *FuncContext*

Provides a *LangLib Func* structure. This is used by components inside a function to determine what function they are a member of.

- *BlockContext*

Provides a *LangLib Block* structure. This is used by components inside a block to determine what block they are a member of.

- *BlockDeadzoneContext*

Provides information about padding required to render the control flow wire overlay in the UI (discussed later).

4.4.4 Pannable and Zoomable Canvas

Implementing a *zoomable user interface* (ZUI) was a significant implementation challenge.

Despite its name, the project *infinite canvas* does not make use of the HTML5 Canvas API[49]. This is because that the Canvas API did not offer enough event handling APIs. If a user clicks on something on a canvas, there is only an event describing *where they clicked*, not *what they clicked*. Implementing such functionality (with large amounts of hierarchical elements) would require using space partitioning techniques such as *quadtrees* or *binary space partitioning*. This behaviour is already largely implemented by the DOM for HTML elements with event bubbling and events for a huge set of behaviours.

The pannable and zoomable canvas consist of regular React components (i.e. HTML and CSS), as does all of its children contained within.

The *UIGraph* component implements the *infinite canvas* and contains the *graph plane*. The *graph plane* is a single *div* element with its CSS *width* and *height* attributes set large enough to make it *pseudo-infinite*, i.e the user will always be able to have the plane within their viewport.

In React, performing state updates in fast intervals can cause poor performance as each state update may cause React to perform a rerender. To overcome this problem, the implementation tracks the *real* offsets for the current pan and zoom (*realPan*, a 2d coordinate and *realZoom*, a scalar) on the graph. When a pan or zoom has finished, the state is updated with the *real* values and a single React rerender occurs rather than a rerender for every single frame during the pan animation.

While the *UIGraph* component is present on the page, a *requestAnimationFrame* loop runs. *requestAnimationFrame* is a function that allows a callback to be called before the next repaint of the page by the browser. It is most commonly used for implementing animations. If *requestAnimationFrame* is not used, animations can appear choppy as CSS attributes are updated mid paint rather than just before a repaint. When called in a loop, it ensures that a function is ran at the browser framerate, usually 60 frames per second. The *requestAnimationFrame* loop in *UIGraph* component updates the *transform* CSS parameter on the *graph plane* to position and scale the plane with the *realPan* and *realZoom* values.

Several event handlers are bound to the *graph plane* element. To implement *zoom towards cursor position* functionality, the *onWheel* (scrollwheel event) is bound. Since the transformation origin of the graph plane is its center, simply increasing or decreasing *realZoom* is not enough. This only implements *orthographic zoom*, not *zoom towards a point* functionality. To fix this, *realPan* must be also be adjusted. The difference in scale is calculated and *realPan* is corrected to zoom towards the mouse position.

The plane has the *draggable* attribute which means that it will fire the *onDrag*{*Start*, *Over*, *Leave*, *End*} events when the corresponding action occurs. The handlers bound to these events only toggle variables that say panning is occurring. A global *mousemove* handler is bound to the window element when the *UIGraph* component is on the page. This is done because the user should

be able to pan the graph for as long as they are holding down their mouse button. A *mousemove* event on the graph plane will stop firing as soon as the user is no longer hovering over the plane.

When panning is occurring (i.e the user is moving their mouse), the global *mousemove* event handler will add the difference (*delta*) of the mouse movement to *realPan*.

All event handlers have *stopPropagation* and *preventDefault* called on the event object, preventing default behaviour from occurring or bubbling up the DOM.

The *UIGraph* component stores anything present on the graph plane as its children. These children can be positioned with *x* and *y* coordinates using the *UIFloating* component described earlier in this chapter. The *GraphContext* provides the current pan, zoom, and graph mouse position to all of these children.

These values are especially important if a child component in the graph plane makes use of mouse-related events that modify positional CSS attributes like *left* or *top*. CSS attributes are used by the browser *pre-scaling* of the graph plane, but a mouse event on an element will produce a mouse position that is *post-scaling* of the graph plane. These child components use the graph zoom value and can call a function called *scale* to rescale the mouse position given by a mouse event.

UIGraph accepts several callbacks as props. These callbacks are bound to the same event handlers on the *graph plane*. The two most significant callbacks used are the *onDrop* callback and the *onClick* callbacks which are required for drag-and-drop functionality and deselecting the current element.

4.4.5 Recursive Components for Blocks, Expressions and Types

Many of the elements in the visual programming language are hierarchical, recursive in nature and form trees. For example, an expression may contain several other expressions, which may contain several more expressions. Attempting to turn these trees into HTML elements iteratively would involve an unnecessarily complex iterative tree-walk approach.

As mentioned in Chapter 3, React was chosen because of its use of

functional components. A sample functional component was shown in subsection 3.1.1.

To solve this challenge, functional components in React were called *recursively* to produce user interfaces with the many required nested elements. Figure 4.4 shows a recursive component in use. Figure 4.5 shows a three-dimensional visualization of the DOM tree from the root component.

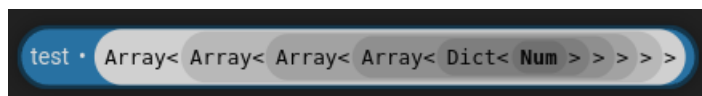


Figure 4.4: *UITypeRepr* component showing the deeply nested type for a variable

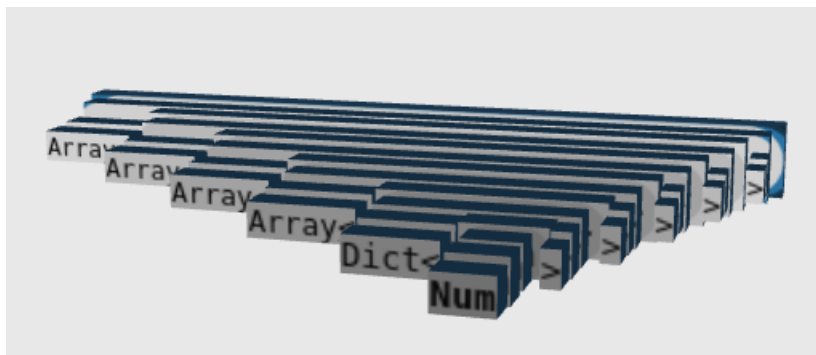


Figure 4.5: *UITypeRepr* recursion visualized in 3D

These recursive components are used to construct the following tree-based elements:

- Blocks (*UIBlock*)
- Expressions (*UIExpr*)
- Type Representation (*UITypeRepr*)

The implementation of these components generally accept the flattened hierarchy *LangLib* uses (generally a hash table with each element in the hierarchy). This hash table is passed as a prop to each child component and an additional prop is passed specifying what identifier in the hash table to use.

For example, *UIExpr* takes an *exprs* property which is a dictionary of all expressions in the hierarchy and an *exprId* property which is the id that the *UIExpr* will represent. Each child within *UIExpr* will either be another *UIExpr* (with the *exprId* being the identifier of the child expression) or a *UIExprParam* if there is no child expression but rather an empty parameter slot.

This raises questions about how to handle events in deeply nested component hierarchies. Listing 4.17 shows the props required for *UIExpr*.

The solution used was to provide callback properties that can reference an expression anywhere in the hierarchy. These can be seen in the listing where several events are defined. These callback functions are passed to each child *UIExpr* component and so on.

Listing 4.17: Props definition of *UIExpr*

```

1 interface UIExprProps {
2   // When an expression anywhere in this
3   // expression tree starts being dragged
4   onExprDragStart?(expr: Expr, ev: React.DragEvent, offset: Vec2): void
5
6   // When an expression anywhere in this expression tree stops being dragged
7   onExprDragEnd?(expr: Expr): void
8
9   // When an expression envelope is attached anywhere in this expression
10  onExprEnvelopeAttached?(
11    droppedOntoExpr: Expr,
12    childId: string,
13    exprEnvelope: ExprEnvelope
14  ): void
15
16  // When an expression anywhere in this expression tree is selected
17  onExprSelected?(node: Expr): void
18
19  // When an expression anywhere in this expression tree is deleted
20  onExprDelete?(node: Expr): void
21
22  // When an expression parameter slot
23  // anywhere in this expression tree is selected
24  onExprParamSlotClick?(parent: Expr, childId: string): void
25
26  // When an expression anywhere in this expression tree is extended
27  onExprExtend?(parent: Expr): void
28
29  // Id to use from flattened hierarchy
30  exprId: string
31
32  // The flattened hierarchy
33  exprs: Record<string, Expr>
34  exprCtx: ExprCtx

```



```

35
36
37   noDelete?: boolean
38   selectedExprId?: string
39   noEvents?: boolean
40   nodrag?: boolean
41   fixed?: boolean
42   border?: string
43   overlay?: boolean
44   mouseLocked?: Vec2
45   typeHighlight?: Type
46   extendable?: boolean
47 }

```

At the root *UIExpr*, the callbacks are implemented to perform the desired behaviour. For example, at the root *UIExpr*, *onExprSelected* can be implemented to call

```
editProjAction({action:"selected",selected:{exprId:selectedExpr.exprId}})
```

This call to *editProjAction* will dispatch the action to the reducer and mutate the *EditProj* structure. This will record that the expression is selected.

If a deeply nested expression is selected, it will simply call `props.onExprSelected(thisExpr)` which will call the *onExprSelected* implemented at the root and cause the deeply nested expression to be selected.

4.4.6 Layouts of Actions, Expressions and Triggers

As discussed in the section exploring *LangLib*, the *NamespaceBuilder* class decouples the visual layout of a language element from the definition and implementation. This was done in the case that the user interface framework may change in the future. It is still important that even when providing layout functionality there is some attempt to make it user interface framework independent. To solve this problem, layouts of actions, expressions and triggers are defined in a *semi-abstracted* form.

The vast majority of layouts will *not* explicitly use React components but rather a abstraction in the form of *helper methods* (that produce React components behind the scenes). If the web framework used by the project were to change, only the helper methods need to be rewritten to support the new framework.

If an action, expression or trigger requires a complex layout it is still free to use React components. This decision was made because abstracting away a highly complex user interface would probably require modifying the

abstractions if the user interface framework were to change. Web frameworks often operate dramatically different from each other. There is no commonly accepted standard, therefore finding a good abstraction for a complex interface is very difficult.

Listing 4.18 shows *helper methods* in use defining the layout for the $+$ expression for the addition of two *Nums*. Figure 4.6 shows the resulting layout. Each helper method generates a React component. More helper methods like *lines* exist for slightly more complex layouts requiring multi-line expressions.

The layout helper methods generally have the same functionality across layout functions for actions, expressions and triggers.

Listing 4.18: Layout helper methods in use with *exprLayout*

```

1 .exprLayout("plus", (ctx: ExprCtx, helper: ExprLayoutHelper) => {
2   const exprDef = ctx.wProj.exprDef(ctx.expr)
3
4   return helper.line([
5     helper.param("a"),
6     helper.text(exprDef.name),
7     helper.param("b")
8   ], 4)
9 })

```

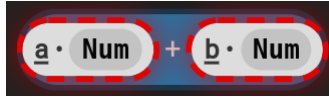


Figure 4.6: $+$ *Num* layout

4.4.7 Drag-and-Drop Support

Drag-and-drop support of expressions and statements is solved via the concept of *envelopes*.

An *envelope* contains an (optional) record of where it has originated from and contains an object of some form.

For example, an *expression envelope* contains the origin project identifier and positioning data of where the expression was originally located. An expression can be floating or inside a parameter slot of a statement or another expression. Alongside the origin will be the flattened expression hierarchy (hash map of expressions) and the expression identifier of the root expression in the hash map.

The *Proj* reducer supports an action named *ExprEnvelopeAttach*. This action can be used to attach the expression stored in the envelope onto anywhere in the project.

Expression envelopes are not required to have an origin. The purpose of the origin is to detect when an expression envelope has originated from the same project it is going to be attached to. When this happens, the handler for the *ExprEnvelopeAttach* action will remove the envelope expressions from their old location and create them at their new location.

If the origin is from a different project or is not specified, all identifiers in the expression are automatically remapped to newly generated identifiers. This is done to prevent any conflicts with existing identifiers in the project.

This implementation means an expression can be dragged from one project and dropped into another.

Envelopes are not exclusively used for drag-and-drop, they can be used for any number of operations. As originless envelopes automatically have identifiers remapped, envelope attach logic can be used to add expressions onto parameter slots or to the infinite canvas without need to implement an *ExprAddAction*.

Statement envelopes operate on the same principle and contain an origin location and record of a statement. For expressions attached to the statement parameter slots, the *statement envelope* contains an *expression envelope* for each parameter.

When a drag of an expression occurs, an expression envelope of the expression is generated. This envelope is then serialized to JSON and the *dataTransfer* attribute of the drag event has its data set to the payload. The *dataTransfer* attribute allows the transfer of a string during a drag until the drop event.

Drops are handled by the *onDrop* handler on the *graph plane* and any parameter slot. The *dataTransfer* object's data is read and deserialized. If an envelope is present in the data, *projAction* is called with the envelope attach action for that specific location.

4.4.8 Control Flow Wires

The rendering and placement of control flow wires was another difficult implementation challenge. This is accomplished via the concept of *deadzones*.

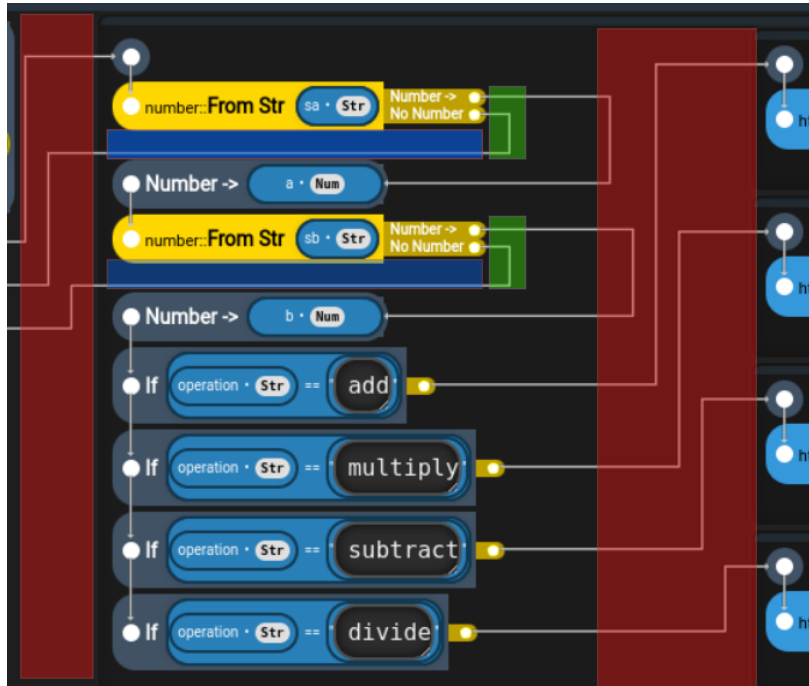


Figure 4.7: Control Flow Wire *deadzones*

These are padded areas of a block that cannot contain statements or any other components. A deadzone provides space for a control flow wire to be overlaid on top of it. An example of these deadzones can be seen in Figure 4.7. There are three types of deadzone in each block. They will be referred to by the color of the deadzone in the figure.

Each deadzone allocates either horizontal or vertical space for a control flow wire. In the figure:

- *Blue deadzones* allocate vertical space for control flow wires travelling horizontally back to the parent block.
- *Green deadzones* allocate horizontal space for control flow wires travelling vertically while on their path to the *blue deadzone*
- *Red deadzones* allocate vertical space for control flow wires travelling within the current block or from/to the child block.

This particular approach eliminates overlapping wires as wires will only ever exclusively intersect horizontally or vertically.

To generate these deadzones, the program graph is traversed. Each jump is categorized into what deadzones it will traverse. Each deadzone has a fixed width for each possible wire that can traverse it. A structure is generated with the widths of each possible deadzone in the program. This structure is then provided by the *BlockDeadzoneContext* mentioned earlier in this chapter. A *UIBlock* component consumes this context and provides the *red deadzone* space as required. A *UIStatement* component also consumes this context to provide the *green* and *blue deadzone* space.

A *UIControlFlowOverlay* component is overlaid on the graph plane and uses SVG to render the wires as *path* elements. A *requestAnimationFrame* loop runs and updates the wire positions as required.

4.4.9 Picker

The *Picker* appears complex but is a relatively unchallenging part of the implementation. The Picker is triggered by the *EditProjPickerAction* and a structure is passed determining where (graph coordinate) and what layout to show. The primary challenge was filtering the information the user can select from.

When requesting elements like structures and expressions, the structure has a record of any namespace or type filters and where that element should be placed.

Filtering and fuzzy search for the Picker is achieved by calls to the *WEditProj* wrapper class.

This class implements methods that performs the fuzzy search for the required element using the *FilterMaps* on both the *EditProj* structure and the *Dimension* structure. It combines the results of the search of the project with the results of the search of the standard library. The Picker filters these results as required. For example, types can be filtered using the type comparison functions in *LangLib*. (e.g. *typeEqualIff*)

The value of the search input is watched by a *useEffect* hook and the filter results are updated as required.

Chapter 5

Evaluation

This chapter will evaluate the implementation produced in Chapter 4 and provide a full assessment of its outcome. Shortcomings and concerns about the implementation will be discussed. Benchmarks are provided to confirm observations about the implementation and ensure requirements defined in Chapter 2 are met.

Every design element defined in Chapter 3 was implemented successfully.

5.1 Open Day Feedback

Feedback from the School of Computer Science and IT FYP Open Day was extremely positive. Several key points were raised by those interested in the project:

- The panning/zooming canvas provides an excellent user experience at quickly navigating around a project.
- The Picker often makes programming faster than a traditional programming language.
- The ability for the Picker to automatically filter expressions based on the type of the currently selected parameter slot is seen as an extremely powerful feature.

- Inlining traditional programming languages like Node.js offers the ability to extend the language features far more than the scope of the standard library.
- The control flow wires can be confusing at times. They are often reminiscent of the same issues caused by *goto* statements in traditional programming languages[7]. “*Visual spaghetti*” is still a problem that needs a solution.
- Highlighting errors by parameter slot rather than line is much more helpful.

5.2 HTTP Benchmarks

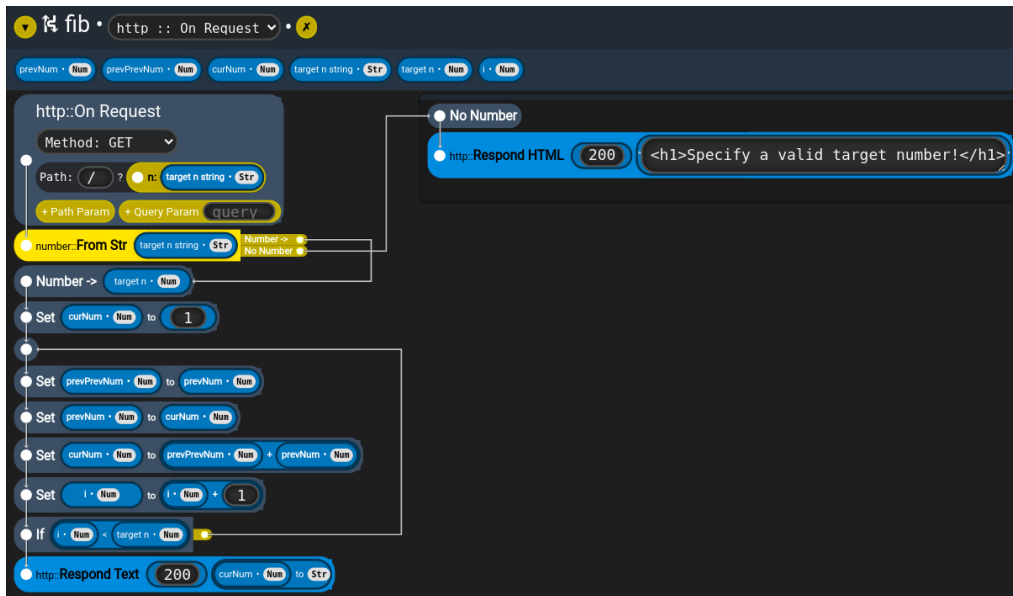


Figure 5.1: Fibonacci Calculator

A simple iterative Fibonacci calculator was constructed. This is shown in Figure 5.1. It parses n as a query parameter, converts it to a number and then writes the n th Fibonacci in response to the request. This project was deployed with a hardcoded memory limit of 75MB.

wrk2[32] was used as the HTTP benchmarking utility. The benchmark was configured to run for 20 seconds with 8 threads, 400 TCP connections and a target throughput of 20,000 requests per second (RPS). The target Fibonacci number was 25. The benchmark was conducted on *localhost* to ensure maximum possible throughput would be achieved.

The benchmark result yielded 1702 RPS but **23525 out of the 34355 requests failed**. This was concerning and was investigated. It was first discovered that the Linux file descriptor limit was being exceeded. A file descriptor is created for each open TCP connection to the service. The file descriptor limit was raised and the benchmark was restarted. This did not solve the issue. The project was not exceeding the memory limit either.

With some investigation of logs it was discovered that the OpenFaaS Gateway (which reverse proxies the request to the local container from *faasd*) chokes heavily on very high RPS loads. A backlog of requests builds up and all requests fail. This was very disappointing.

Further benchmarks were carried out and it was discovered that the OpenFaaS gateway can handle approximately 600 - 1000 RPS before problems manifest.

600 - 1000 RPS can be viewed as meeting the project requirements but further investigation is required.

The result does not test the true potential performance of what the language can achieve due to the artificial bottleneck. An alternative benchmark was carried out to investigate this. The container image of the project was pulled from the registry and ran externally with Docker (i.e, not managed by OpenFaaS). The same *wrk2* benchmark was carried out and the results were dramatically different.

The project handled **14325 RPS over 20 seconds**. 286705 requests were handled in this period. This was a massive improvement over the previous experiment. This benchmark demonstrated that the OpenFaaS Gateway is a massive bottleneck for performance.

5.3 Deployment Stage Benchmarks

As a considerable amount of effort was spent trying to reduce deployment time, benchmarks are crucial to determine if this made an observable difference.

Deployment Stage	Uncached (s)	Cached (s)	Improvement %
Transpile	0.01	0.01	-
Write Deployment Files	0.02	0.02	-
Tar Deployment Files	0.03	0.03	-
Upload and Build Image	6.61	2.42	173%
Push Image	5.69	1.04	447%
Deploy Function	12.52	11.13	12%
Total	24.88	14.65	70%

Table 5.1: Deployment Stage Benchmarks.

The Docker build cache was cleared and a simple calculator project was deployed. The time taken by each deployment stage was recorded. The experiment was repeated 25 times to collect values for a mean.

These steps were then taken with the build cache in use. A single property on the project was modified to invalidate the cache for the transpilation build stage. This is the most likely situation a user will encounter when performing deployments during editing.

Table 5.1 shows the results of the benchmarks. The time of each deployment stage is shown.

As expected, the build cache offers a dramatic improvement during the Docker image build. A performance improvement was expected in the push stage but not to the extent of the values discovered. A large amount of time is saved in this stage when the build is cached because the base layers of the image are already present on the registry. This drastically reduces upload time as only a few small layers need to be pushed.

The Deploy Function time (which relies on OpenFaaS) was extremely disappointing. A small improvement was observed but a larger improvement was expected. Layers of the build image are expected to be cached on the host running OpenFaaS. This means that less images need to be pulled and downloaded from the registry. Once again, OpenFaaS is observed to be a bottleneck. Theoretically, the build files are available by the end of the *Upload and Build Image* stage. This could mean with additional development effort, it may be possible to deploy a user project (with build caching enabled) within 2.48 seconds. This would require eliminating the dependency on OpenFaaS

and implementing a new alternative.

Overall, a 70% improvement in deploy time was observed if build caching was enabled. This is an acceptable result.

5.4 Requirements Checklist

The list of requirements elicited in Chapter 2 is shown. The source of how the requirement has been met has been added.

1. The platform must allow the user to create separate workspaces.

Project System

2. The language must be able to handle HTTP requests (query parameters, path matching, body) and respond as necessary.

HTTP Trigger, HTTP Actions

3. The language must be optimized for coding rapidly and at speed.

Picker UI, Pannable Graph

4. The language must not be proprietary and force a user into a walled garden.

Inlined Traditional Programming Languages

5. The language API must be easily extendable.

NamespaceBuilder

6. The language must feature an IntelliSense or hinting system.

Picker UI, FilterMaps, Fuzzy-search

7. The language must feature a type system comparable to a traditional programming language.

Variables, JSON Inspired Type System

8. The language must feature a data definition system.

Model System

9. The language should make an attempt at preventing syntax errors.

Expression Drag Type Checking

10. The language should not confuse a user.

Open Day Feedback

11. A project must execute securely alongside other projects.

OpenFaaS, runc Container Isolation

12. The runtime must allow a basic project to withstand a minimum of 500RPS (requests per second).

HTTP Benchmark

13. The runtime must allow a basic project to use less than 75MB of memory.

HTTP Benchmark

14. A basic project must deploy in less than a minute.

Deployment Benchmark

5.5 Scalability Concerns and Future Work

While the artefact was successfully constructed, there exists several concerns about the implemented system. Issues from these concerns will manifest as the system is used by more and more individuals. These concerns manifest the need for possible future work on the project.

5.5.1 Standard Library Additions

The standard library is missing functionality that is expected of other programming languages. Features that are missing include sorting algorithms, random number generation, and math functions. Until steps are taken to implement these, the language will not be seen seriously by both non-programmers and regular programmers alike.

5.5.2 Tables and Queues

There is no concept of data storage in the language. A new feature could allow the user to create a table with fields for storing data. The table row would be usable as a model type. A table could be composed of many other existing tables (which would allow the implementation of joins). The user would be able to see, query and edit the table contents in the canvas editor. A new set of expressions that describe predicates would allow the user to query the table.

There is a need for a queueing system. This would allow the user to manage several long-running and complex tasks by pushing data to a queue. A queue would only be able to carry model instances of particular types. The user would be able to write functions with triggers that relate to the queue. For example, a trigger could fire if the queue was not empty. This would allow the user to pop the value from the queue and process it.

5.5.3 User Accounts

There is no concept of user accounts implemented. Projects are simply stored in the database and not tied to any particular user. The system is not truly *multi-tenant* as most development focus was on the design and implementation of the visual programming language itself.

5.5.4 API Security

As the FYP was implemented without the concept of user accounts, authentication is missing from calls to the backend from the frontend. *ProjSvc* does not currently implement any security for any calls made over JSON-RPC. Any client can arbitrarily send a payload to the web server and perform actions.

RPCs would need to be implemented using authentication tokens, this would involve supporting the *OAuth2 specification* or *JSON Web Tokens* (JWTs)

5.5.5 Real Time Editing

Currently a project can only be edited by a single user at a time, there is no *multiplayer* functionality. This means the project is not truly collaborative. Implementing multiplayer functionality is extremely complex and would rely on data structures known as *conflict-free replicated data types* (CRDTs)[75].

Clients would be expected to communicate over a real-time link such as a WebSocket connection. It is expected that connections may be poor and clients may go offline while editing. This needs to be accounted for.

CRDTs are data structures that can be modified independently by several clients. A client can be completely offline and make a modification to the CRDT. When clients come online and sync, CRDTs can automatically resolve merges and conflicts.

This is the core technology behind real-time collaborative editing functionality as seen with online editors such as Google Docs

5.5.6 Visual Debugger

There is no *visual debugger* offered to the user. There is no way for a user to trace an execution in progress and step statement by statement. There is no means to visualize the value of a variable of an in-progress request. This would involve developing a debug protocol for communicating debugging information from server to client.

5.5.7 Instrumentation and Observability

The only way of instrumenting a running project is with a *health check*. This returns a *200 OK HTTP* response if the server process inside the project container is running.

There is no fine-grained instrumentation or *observability* features built into the system. This means that records of requests to the project are unknown. Ideally, a system would be implemented that can monitor a running project and log any requests to a database. Logging CPU, memory and I/O usage would provide the user with helpful ways of seeing that a particular operation is causing problems.

Queries from this database would then be able to be made from the user interface. Tracing the execution path of a particular request on a project

(via highlighting statements and control flow wires) would be an interesting aspect to explore in the future.

5.5.8 Multi-Node Build Caching

The Docker build cache is restricted to the host the Docker daemon is installed on.

If multiple Docker daemons are in use, they will all have an independent build cache. There is currently no mechanism for easily sharing the cache across a fleet of hosts.

To solve this issue, a *build service* would need to be developed that sat in front of all container image builds. It would need to track what daemons cache what particular layers and prioritize particular builds with particular daemons. Another alternative is to store cached builds on a distributed filesystem such as *Ceph* or *GlusterFS* so they are accessible on every single Docker build daemon.

5.5.9 Multi-Node OpenFaaS

faasd (the OpenFaaS deployment) is restricted to a single node. It does not offer a clustering mechanism. This means that the current implementation only deploys to a single server. Therefore the maximum number of running projects is restricted to the resources the server is able to provide.

This could be alleviated by running *OpenFaaS* on a Kubernetes cluster which can easily scale the number of worker nodes and distribute containers across the cluster.

The problems discovered relating to OpenFaaS during the evaluation give cause for a custom serverless engine to be written exclusively for the project. This would reduce the reliance on an external dependency, may lead to better performance and can be tightly integrated into the core project.

5.5.10 Container Isolation Vulnerabilities

The container tools *containerd/runc* are used by OpenFaaS for isolating projects as they can contain untrusted code. An isolated container still has a very large attack surface. Container escape attacks use vulnerabilities present

in both the container runtime *runc* (which sets up the container) and the Linux kernel. *CVE-2019-5736* was a full container breakout vulnerability in *runc* that allowed for total compromise of the host system[5][93].

Dirty Pipe (CVE-2022-0847) was a vulnerability discovered in the Linux kernel that allowed for full privilege escalation. Not only did it allow for this from unprivileged processes, but it could also be exploited from within containers[39][3].

While these exploits were patched quickly, the frequency of new vulnerabilities is concerning. A single vulnerability resulting in a compromise would allow an attacker to access all other projects on the system.

In general, the solution to this problem is to use isolation and virtualization with a smaller attack surface. This would mean moving to virtual machines (VMs) and using a hypervisor such as KVM. This would offer more security but at the trade-off of being much more resource intensive.

Chapter 6

Conclusion

As discussed during the evaluation, the project meets the requirements and was well received during Open Day demonstrations. Future work was also discussed from the perspective of solving any issues that came to light with the project.

6.1 Reflection

Prior to the FYP, I felt like I had quite poor skills with algorithms and data structures. Building *LangLib* forced me to improve them as I spent a significant amount of time working with linked-lists, trees and graphs. I spent a lot of time working with recursion and various traversals of data structures. I now feel like I know them like the back of my hand.

In the past when I worked on projects I didn't stick to any kind of implementation methodology. During the FYP I wanted to focus on using structured design patterns and I can now appreciate their use and how they can improve the design of an application.

I got to work with nearly every aspect of a software development stack. I got to build a frontend and backend, work with networking tools and Linux features like containers. I got to work on my Computer Science fundamentals and I got a chance to play around with a huge amount of free open-source software. Working on the report finally gave me a chance to learn and use L^AT_EX. It was a fun-filled adventure across 6 months.

The final outcome is that I built something that I'm proud of and happy

to show off. I feel like the final artefact can truly demonstrate my ability and skills.

6.2 Project Timeline

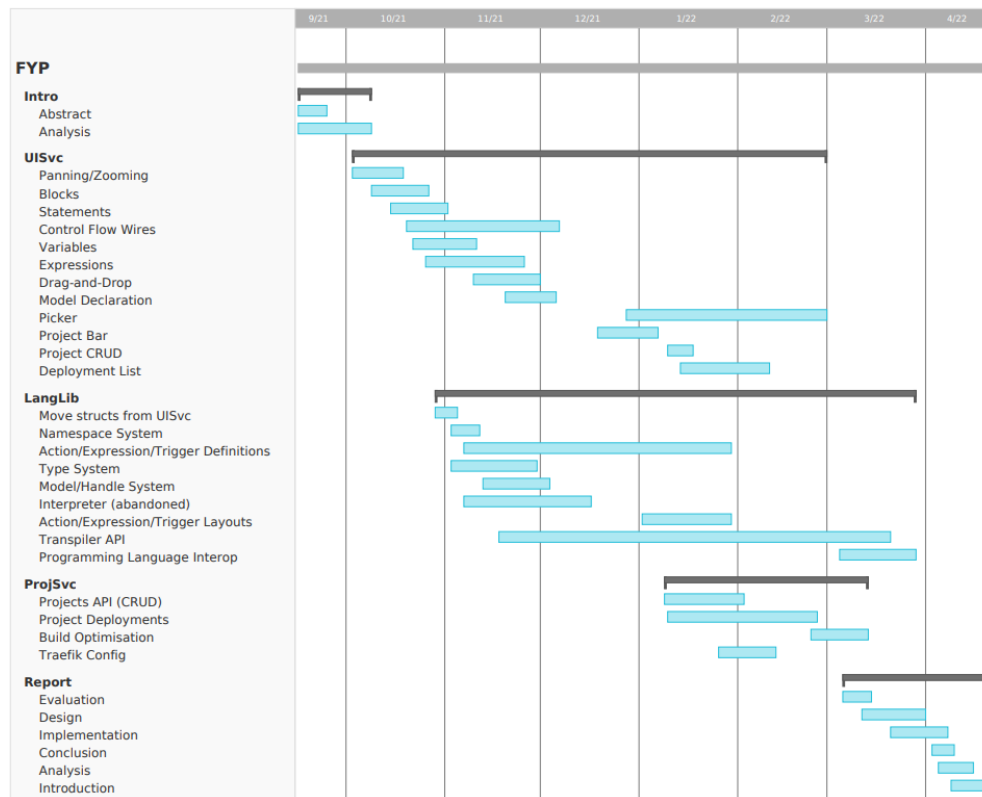


Figure 6.1: Gantt Chart

A rough project *Gantt* chart was used at the start of the project to plan out the project timeline. Figure 6.1 shows a corrected Gantt chart with the full project timeline. A *Kanban* with columns of *To Do*, *In Progress*, and *Done* was used to track implementation tasks.

From November 1st 2021, *WakaTime*[1] was used to track the amount of time spent on development. WakaTime is a dashboard and extension for Visual Studio Code that uploads information about the file currently being worked on. Time is only tracked when the user is adding or removing content

527 hrs 39 mins from Mon Nov 1st 2021 until Today

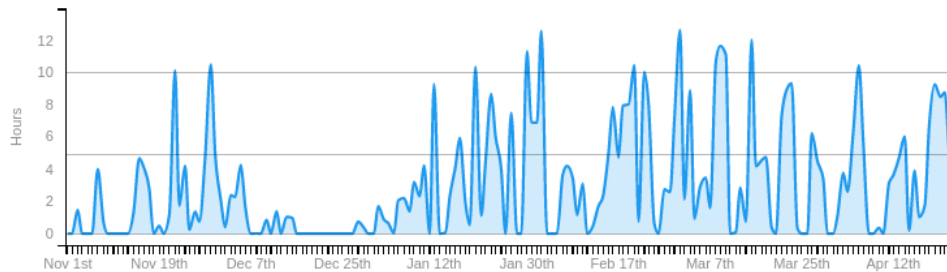


Figure 6.2: WakaTime Graph

from the file, simply having the file open will not add to the timer. As a result, WakaTime is an excellent tool at measuring productivity

As seen in Figure 6.2, approximately **527 hours** were spent on the project.

Of these hours, ~15% of time was spent working on the report (~**79 hours**). ~85% of time was spent working on development (~**449 hours**).

Development time can be categorized into **36 hours** (*ProjSvc*), **143 hours** (*LangLib*) and **270 hours** (*UISvc*). It is likely the true values are higher as data collection only began from November 1st.

The implementation first began on *UISvc*, the visual aspect of the project. This was done to inform the implementation decisions that would need to be made for *LangLib*. From there *UISvc* and *LangLib* were implemented simulatenously. *ProjSvc* began its implementation about half way through the implementation of *LangLib*. Work continued until the completion of all services.

Work on the report started on the 7th March and continued until the project submission date.

Bibliography

- [1] Alan Hamlett. *WakaTime - Dashboards for developers*. URL: wakatime.com/
- [2] C++ Reference Contributors. *C++ Reference - Lambda expressions* URL: en.cppreference.com/w/cpp/language/lambda
- [3] Christophe Tafani-Dereeper. et al. *Using the Dirty Pipe Vulnerability to Break Out from Containers*. March 2022. DataDog Engineering Blog. URL: www.datadoghq.com/blog/engineering/dirty-pipe-container-escape-poc/
- [4] Crunchtools. *A Comparison Of Linux Container Images*. June 2020. crunchtools.com/comparison-linux-container-images/
- [5] *CVE-2019-5736* URL: www.cvedetails.com/cve/CVE-2019-5736
- [6] Devraj Singh. *Why Go compiles so fast*. January 2021. URL: devrajcoder.medium.com/why-go-compiles-so-fast-772435b6bd86
- [7] Dijkstra, Edsger W. *Letters to the editor: Go to statement considered harmful*. March 1968. Communications of the ACM
- [8] Docker, Inc. *Docker*. URL: www.docker.com/
- [9] Docker, Inc. *Dockerfile reference*. URL: docs.docker.com/engine/reference/builder/
- [10] Docker, Inc. *Use multi-stage builds*. URL: docs.docker.com/develop/develop-images/multistage-build/
- [11] Docker, Inc. *Docker Hub Image Registry*. URL: hub.docker.com/
- [12] Drakon.Tech. *DRAKON-Javascript* URL: drakon.tech/

- [13] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides *Design Patterns: Elements of Reusable Object-Oriented Software*. 1994. Addison Wesley. ISBN 0-201-63361-2.
- [14] ECMAScript Authors *ECMAScript® 2023 Language Specification* URL: tc39.es/ecma262/
- [15] Epic Games, Inc. *Unreal Engine Documentation - Blueprint* URL: docs.unrealengine.com/4.27/en-US/ProgrammingAndScripting/Blueprints/TechnicalGuide/Compiler/
- [16] Epic Games, Inc. *UnrealScript Reference* URL: docs.unrealengine.com/udk/Three/UnrealScriptReference.html
- [17] Evan You. *Vue 3 as the New Default*. 2022. URL: blog.vuejs.org/posts/vue-3-as-the-new-default.html
- [18] F5 Inc. *Controlling NGINX Processes at Runtime* URL: docs.nginx.com/nginx/admin-guide/basic-functionality/runtime-control/
- [19] Facebook Open Source. *Flux* URL: github.com/facebook/flux/tree/main/examples/flux-concepts
- [20] Facebook Open Source. *React - Using the State Hook*. URL: reactjs.org/docs/hooks-state.html
- [21] Facebook Open Source. *Function and Class Components in React*. 2018. URL: reactjs.org/docs/components-and-props.html
- [22] Feng Xiao (Google). *Protocol Buffers GitHub Issue - why message type remove 'required, optional'? [sic]* 2016. URL: github.com/protocolbuffers/protobuf/issues/2497#issuecomment-267422550
- [23] Free Software Foundation. *namespaces(7)*. URL: man7.org/linux/man-pages/man7/namespaces.7.html
- [24] Free Software Foundation. *unshare(2)*. URL: man7.org/linux/man-pages/man2/unshare.2.html
- [25] Free Software Foundation. *cgroups(7)* URL: man7.org/linux/man-pages/man7/cgroups.7.html
- [26] Free Software Foundation. *capabilities(7)* URL: man7.org/linux/man-pages/man7/capabilities.7.html

- [27] Free Software Foundation. *seccomp(2)*. URL: man7.org/linux/man-pages/man2/seccomp.2.html
- [28] Google, Inc. *gRPC - A high performance, open source universal RPC framework* URL: grpc.io/
- [29] Google Inc, Go Contributors. *net/http GoDoc*. URL: pkg.go.dev/net/http
- [30] Google, Inc. *Protocol Buffers* URL: developers.google.com/protocol-buffers
- [31] Google, Inc. *V8 JavaScript engine* URL: v8.dev/
- [32] Gil Tene. *wrk2 - A constant throughput, correct latency recording variant of wrk*. URL: github.com/giltene/wrk2
- [33] Green, Thomas & Petre, Marian. *Figure 11: Visual spaghetti from Usability Analysis of Visual Programming Environments: A 'Cognitive Dimensions' Framework* 1996.
- [34] IEEE. *IEEE Standard for Floating-Point Arithmetic. IEEE STD 754-2019* doi:10.1109/IEEESTD.2019.8766229. ISBN 978-1-5044-5924-2. IEEE Std 754-2019.
- [35] Ierusalimschy, et al. *The Implementation of Lua 5.0* URL: www.lua.org/doc/jucs05.pdf
- [36] Joe Duffy. *Concurrency Hazards: Solving Problems in Your Multi-Threaded Code*. Microsoft. 2008. URL: docs.microsoft.com/en-us/archive/msdn-magazine/2008/october/concurrency-hazards-solving-problems-in-your-multithreaded-code
- [37] JSON-RPC Working Group. *JSON-RPC - A light weight remote procedure call protocol*. URL: www.jsonrpc.org
- [38] Let's Encrypt Foundation. *Let's Encrypt* URL: letsencrypt.org/
- [39] Max Kellerman *The Dirty Pipe Vulnerability* February 2022. URL: [dirtypipe.cm4all.com/](https://cm4all.com/)
- [40] Microsoft. *Visual Studio Code Documentation - IntelliSense* URL: code.visualstudio.com/docs/editor/intellisense

- [41] Microsoft. *REST API Guidelines* URL: github.com/microsoft/api-guidelines/blob/vNext/Guidelines.md
- [42] Microsoft. *TypeScript Language* URL: www.typescriptlang.org
- [43] Microsoft. *Windows API Programming Reference - About Handles and Objects*. July 2021. URL: docs.microsoft.com/en-us/windows/win32/sysinfo/about-handles-and-objects
- [44] MinIO, Inc. *MinIO Object Storage* URL: min.io/
- [45] Mike Hadlow. *Visual Programming - Why It's a Bad Idea*. October 2018. URL: mikehadlow.blogspot.com/2018/10/visual-programming-why-its-bad-idea.html
- [46] Miles Berry. *Visual Programming*. URL: milesberry.net/2018/11/visual-programming/
- [47] Milner, Robin A *Theory of Type Polymorphism in Programming*. 1978. Journal of Computer and System Sciences. doi:10.1016/0022-0000(78)90014-4.
- [48] Mozilla Contributors. *AbortController API* MDN. URL: developer.mozilla.org/en-US/docs/Web/API/AbortController
- [49] Mozilla Contributors. *Canvas API* MDN. URL: developer.mozilla.org/en-US/docs/Web/API/Canvas_API
- [50] Mozilla Contributors. *Canvas Element (Maximum canvas size)* MDN. URL: developer.mozilla.org/en-US/docs/Web/HTML/Element/canvas#maximum_canvas_size
- [51] Mozilla Contributors. *Document Object Model* MDN. URL: developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model/Introduction
- [52] Mozilla Contributors. *Single Page Application* MDN. URL: developer.mozilla.org/en-US/docs/Glossary/SPA
- [53] Mozilla Contributors. *Web APIs* MDN. URL: developer.mozilla.org/en-US/docs/Web/API
- [54] Mozilla Contributors. *WebSocket API* MDN. URL: developer.mozilla.org/en-US/docs/Web/API/WebSockets_API

- [55] Oracle, Inc. *The Java® Virtual Machine Specification* URL: docs.oracle.com/javase/specs/jvms/se7/html/
- [56] Open Container Initiative (OCI). *runc Container Specification v1*. 2018. URL: github.com/opencontainers/runc/blob/main/libcontainer/SPEC.md
- [57] Open Container Initiative (OCI). *OCI Image Format* URL: github.com/opencontainers/image-spec
- [58] Open Container Initiative (OCI). *OCI Image Format - Configuration* October 2021. URL: github.com/opencontainers/image-spec/blob/main/config.md
- [59] Open Container Initiative (OCI). *containerd default seccomp profile*. URL: github.com/containerd/containerd/blob/eaf286224b0144dd11c0e68c131d5dd9ebf52a23/contrib/seccomp/seccomp_default.go#L52
- [60] OpenFaaS Author(s) *OpenFaaS*. 2021. URL: www.openfaas.com/
- [61] OpenFaaS Author(s) *OpenFaaS - faasd* 2021. URL: docs.openfaas.com/deployment/faasd/
- [62] OpenJS Foundation. *Node.js HTTP API*. URL: nodejs.org/api/http.html
- [63] OpenJS Foundation. *The Node.js Event Loop*. URL: nodejs.org/en/docs/guides/event-loop-timers-and-nexttick/
- [64] Pereira. et al. *Energy Efficiency across Programming Languages* URL: <https://greenlab.di.uminho.pt/wp-content/uploads/2017/10/sleFinal.pdf>
- [65] PostgreSQL Contributors. *PostgreSQL - JSON Types* URL: www.postgresql.org/docs/current/datatype-json.html
- [66] P. Leach (Microsoft). *UUID Specification*. RFC4122 URL: datatracker.ietf.org/doc/html/rfc4122
- [67] ReduxJS Contributors. *Redux - State Actions Reducers* URL: <https://redux.js.org/tutorials/fundamentals/part-3-state-actions-reducers>
- [68] ReduxJS Contributors. *Redux - Three Principles* URL: <https://redux.js.org/understanding/thinking-in-redux/three-principles#changes-are-made-with-pure-functions>

- [69] ReduxJS Contributors. *Redux - Reducing Boilerplate* URL: redux.js.org/usage/reducing-boilerplate#action-creators
- [70] Robert Nystrom. *Crafting Interpreters - A Tree-Walk Interpreter* URL: craftinginterpreters.com/a-tree-walk-interpreter.html
- [71] Riteek Srivastav. *A complete journey with Goroutines* URL: riteeksrivastava.medium.com/a-complete-journey-with-goroutines-8472630c7f5c
- [72] Scratch. *Scratch 3.0 Virtual Machine*. URL: github.com/LLK/scratch-vm
- [73] Samuel Groß *JITSploitation I: A JIT Bug*. Google, Inc. September 2020. URL: googleprojectzero.blogspot.com/2020/09/jitsploitation-one.html
- [74] Seva Safris. *A Deep Look at JSON vs. XML, Part 1: The History of Each Standard*. July 2019. URL: www.toptal.com/web/json-vs-xml-part-1
- [75] Shapiro, Marc. et al. *Conflict-Free Replicated Data Types*. Stabilization, Safety, and Security of Distributed Systems.
- [76] TechEmpower. *Web Framework Benchmarks* URL: www.techempower.com/benchmarks/#section=data-r20&hw=ph&test=db
- [77] T. Bray, Ed. *Javascript Object Notation*. RFC8259. URL: datatracker.ietf.org/doc/html/rfc8259
- [78] The GraphQL Foundation. *GraphQL - A query language for your API* [<https://graphql.org/>]
- [79] The Linux Foundation. *cgroup freezer* URL: www.kernel.org/doc/Documentation/cgroup-v1/freezer-subsystem.txt
- [80] Tiago Simões. *Visual Programming Is Unbelievable* URL: www.outsystems.com/blog/posts/visual-programming-language/
- [81] T. Berners-Lee. *Uniform Resource Identifier*. RFC3986 URL: datatracker.ietf.org/doc/html/rfc3986
- [82] Torstein K. Johansen. *Improve Tomcat Startup Time*. URL: skybert.net/java/improve-tomcat-startup-time/
- [83] Traefik Labs. *Traefik - Configuration Discovery* URL: doc.traefik.io/traefik/providers/overview/

- [84] Traefik Labs. *Traefik - Let's Encrypt* URL: doc.traefik.io/traefik/https/acme/
- [85] Traefik Labs. *Traefik - HTTP Provider* URL: doc.traefik.io/traefik/providers/http/
- [86] ULID Contributors. *The canonical spec for ulid*. URL: github.com/ulid/spec
- [87] styled-components Contributors. *styled-components library* URL: styled-components.com/
- [88] Vladislav Supalov. *Docker - How Does the Docker Cache Work?* 2022. URL: vsupalov.com/docker-cache/
- [89] Victor (Tenthousandmeters). *Python behind the scenes 1: how the CPython VM works*. URL: tenthousandmeters.com/blog/python-behind-the-scenes-1-how-the-cpython-vm-works/
- [90] Viktor P. Ivannikov. *Visual Syntax of the DRAKON Language* Official English Translation of Programmirovaniye. 1995 URL: drakon.su/_media/video_i_prezentacii/graphical_syntax_.pdf
- [91] Wright. et al. *Large-Scale Automated Refactoring Using ClangMR* 2013. Proceedings of the 29th International Conference on Software Maintenance.
- [92] Wix Engineering. *Developing Large-Scale Applications with TypeScript* June 2018. URL: www.wix.engineering/post/developing-large-scale-applications-with-typescript
- [93] Yuval Avrahami. *Breaking out of Docker via runC. Explaining CVE-2019-5736* Palo Alto Networks URL: unit42.paloaltonetworks.com/breaking-docker-via-runc-explaining-cve-2019-5736/
- [94] Sutherland. *Sketchpad: A man-machine graphical communication system* Technical Report Number 574. 2003.