MF: A Web Front-End for Mailboxer

Zubair Khalid

School of Computing Science
Sir Alwyn Williams Building
University of Glasgow
G12 8QQ

A dissertation presented in part fulfilment of the requirements of the Degree of Master of Science at The University of Glasgow

10th September 2025

Abstract

Actor languages, like Erlang, are effective for engineering scalable distributed systems, and exploit asynchronous asynchronous message passing. Like other distributed technologies, actor systems are vulnerable to communication errors like unexpected messages. Traditional type systems cannot guarantee communication correctness. Mailbox types are a new behavioural typing discipline that address this gap by describing the unordered and selective contents of actor mailboxes. Mailboxer is a research tool that has been developed to statically detect Erlang communication errors, i.e. at compilation time. Currently Mailboxer is only usable from the command-line and requires the installation of a complex software stack, including OCaml and the Z3 theorem prover.

This project designs, implements, and evaluates MF, a web front-end that makes Mailboxer accessible in the browser, and teaches the underlying behavioural typing concepts needed through a guided learning journey. MF contributes: (1) a structured literature review and analysis of research in type systems, behavioural typing, the actor model, and mailbox types, (2) a learning-journey design with wireframed pages to introduce actor communication errors, behavioural types, and Mailboxer, followed by example programs of Mailboxer errors (3) an implementation of an interactive sandbox which integrates a Dockerised Mailboxer with a React-based front-end, Monaco editor for the sandbox utility, and an Express.js bridge for communication between containers to enable users to run and modify mailbox-typed Erlang code without local setup, (4) an evaluation comprising of functionality tests, cross-platform containerised deployment, performance measurements, and a user study with 11 participants.

MF's two-container architecture (website + Mailboxer) is a lightweight (50-80MB RAM usage) portable (Windows/macOS/Linux), and responsive (sub-second sandbox analysis). Across the predefined examples (5 correct, 5 with errors, 1 challenge), the sandbox outputs replicate Mailboxer running natively. The user study results conclude high usability (navigability 9.5/10, and sandbox clarity 9/10) and strong knowledge gains (e.g. 91% correctly identify payload mismatches and 100% recall when mailbox types were first introduced). These results show that MF lowers the barrier to exploring mailbox types and offers a simple path from behavioural type theory toward practice in an educational context. Rather than MF offering direct improvements to real-world reliability, it positions itself as a learning aid that may help users better understand communication correctness in actor-based concurrent systems.

Education Use Consent

I hereby give my permission for this project to be shown to other University of Glasgow students and to be distributed in an electronic format. **Please note that you are under no obligation to sign this declaration, but doing so would help future students.**

Name: Zubair Khalid Signature: Zubair Khalid

Acknowledgements

I would like to thank my supervisor, Professor Phil Trinder, for his constructive feedback, guidance, and for his support throughout this project. Without his mentorship, I would not have been able to perform to the best of my ability.

I am also grateful to Robert Law, my supervisor for my undergraduate honours project at the Glasgow Caledonian University. He gave me the motivation and confidence to apply for the MSc in Computer Science at the University of Glasgow.

I am thankful to the University for allowing me the privilege and opportunity of studying here for the duration of my MSc.

I am especially grateful to James Moonan and Sean McGrevey, my high school Computer Science teachers who pushed me, helped me unlock my potential, and nurtured my love for programming.

Most importantly, I am thankful to my friends, and above all, my family. My parents, Zaqaria and Khalid, and my siblings, Iqra, Jahanzab, and Umair. Your love and support throughout this past year have kept me motivated and helped me succeed.

Contents

Chapter 1: Introduction to MF

1.1 Overview

Actor-based systems such as Erlang enable the engineering of reliable and scalable distributed systems by providing asynchronous message passing and isolated state [1]. This avoids common errors of shared-state concurrency, including race conditions, dead-locks, and live-locks. However, like other distributed paradigms, actors are vulnerable to communication errors such as mismatched payloads, omitted replies, and unexpected requests. While Erlang is dynamically and strongly typed, allowing the ability to rapidly develop concurrent systems, it does not enforce guarantees about communication correctness.

Traditional data type systems ensure functional/API and data correctness but cannot reason about communication protocols. Behavioural typing [2] extends the scope of type systems to cover communication protocols. In the context of actors, mailbox types [3] use commutative regular expressions to describe the possible multi-set of messages that may reside in a mailbox, capturing the unordered and selective nature of actor communication.

`Mailboxer` [4] addresses this by enabling Erlang code to be type-checked against mailbox types, via translation into the Pat [5] language. However, its reliance on command-line execution and environment-specific setup, which includes OCaml, the Z3 theorem prover, and the Erlang/OTP, creates a barrier for its usage. This project addresses these limitations by introducing `MF`: a web front-end for Mailboxer, that makes the tool accessible within a browser environment. MF is designed as a guided learning platform that introduces key concepts, such as actor communication errors, behavioural types, and mailbox types, before allowing users to directly experiment with Mailboxer in an interactive sandbox.

1.2 Research Contributions

This project makes the following research contributions:

1. **Background Research and Literature Review**
   A structured review and analysis of existing research in type systems, behavioural typing, the actor model, and mailbox types [3] [5] is presented. This review establishes the theoretical foundation required to understand the motivation behind the development of Mailboxer [4] - an Erlang [6] to Pat transpiler to provide type-checking to Erlang - and it's potential to improve the reliability of actor-based systems. The literature review covers type systems - static and dynamic, to behavioural typing and session typing - to highlight their strengths and limitations in concurrent and distributed systems. Attention is given to the the mismatch between session types and the actor model, which motivates the development of mailbox types as a better fit for asynchronous and unordered communication found in actor languages. The aim of this project is to develop a web front-end for Mailboxer (MF), to provide an interactive and guided learning experience from theory to practice, with the additional benefit of being able to experiment with mailbox types within an actor based language, such as Erlang. This is to make the mailbox typing developed in the STARDUST [7] project accessible to a wider audience of developers. This contribution allows the design and implementation of MF to be grounded in clear understanding of the underlying theory, and its practical implications. Found in Chapter 2, Sections 2.1-2.10, and Appendix A.

2. **The Design of MF: Web Front-End for Mailboxer**
   Wireframes, created using Balsamiq [8], are utilised to plan the navigation flow and page structure - to ensure consistency and clarity. The design follows a learning-journey approach, where users are guided from theory to practical examples i.e. Erlang programs that contain communication errors, ensuring complex theory is presented as approachable to students and current developers. The design structured around dedicated sections for explaining actor communication errors, behavioural types, and Mailboxer itself (a tool to transpile Erlang to Pat, a programming language design incorporating mailbox types) - with additional supporting material and examples. The design is intended to make advanced behavioural typing research concepts more accessible and understanding to the developer community, to bridge the gap between theoretical work on mailbox types and allow actor language developers to experiment with a practical application of mailbox types via Mailboxer. Details in Chapter 3, Sections 3.1-3.4, and Appendix B.

3. **Implementation of MF: A Browser-Based Sandbox for Mailboxer**
   The main technical contribution of this project is the development of an interactive browser-based sandbox, that integrates directly with Mailboxer. This feature allows users to use Mailboxer to experiment with communication errors in Erlang programs without needing to install a complex software stack. The sandbox is implemented using Monaco code editor [9] to provide the browser-based code editor, with syntax highlighting. The server, utilising Express.js [10], bridges the communication between the front-end and the Mailboxer container. User code, written in Erlang, is passed to the Mailboxer container, which its compiled binaries are executed upon it by first transpiling to Pat - as part of the mailbox type checking, Pat calls the Z3 theorem prover to solve constraints - with the results returned, parsed, and displayed to the user of MF, all in real time.

   The website itself contains of 10 pages, ranging from the home page (entry point) to the sandbox (web front-end for Mailboxer). Each page is designed to be a consistent

layout, with key information pertaining to the topics of that page in a format that is both digestible and easy-to-understand for the uninitiated.

The sandbox supports several methods of interaction. From 5 working mailbox-annotated Erlang examples, with the inclusion of 5 examples containing errors, and a challenge exercise - for users to apply their understanding of core concepts defined in prior sections of MF, to correct the code. This contribution demonstrates how advanced research tools such as Mailboxer, which can be difficult to install and use, can be made accessible through a lightweight browser-based interface. This is achieved through containerisation of the Mailboxer tool - originally not distributed in a portable or easy-to-deploy form.

A custom Dockerfile [11] packages Mailboxer and its dependencies into a lightweight container, allowing it to run consistently on different environments. The system architecture is designed as a two-container stack: one for the React-based front-end and Express.js server, and the second for Mailboxer itself. The server communicates with Mailboxer via `docker exec`, which allows user-edited code from the sandbox to be analysed in isolation. This separation of concerns ensures maintainability, portability, and security, while keeping resource usage low. By providing a Dockerised version of Mailboxer, this project extends the usability of the research tool, making it to be accessible to the wider developer community.

The final system is developed using React [12], Vite [13], `react-router-dom` [14], Bootstrap [15], Highlight.js [16], and @monaco-editor/react [17]. This results in MF being structured as a single-page application, with the implied experience of a multi-page application. `react-router-dom` is used to reduce perceived load time and improve the user experience. The implementation contains 3K Source Lines of Code (SLOC), which spans multiple technologies: TypeScript/JavaScript (front-end and back-end), Erlang (annotated examples), and Docker (containerisation), which is detailed in Chapter 4, Sections 4.1-4.4, and Appendix C.

4. **Evaluation of MF: Functionality, Portability, and User Study**
The final contribution of this project is the evaluation of MF, which is done by functionality testing, cross-platform deployment, and a user study. Functionality testing is done to verify if the website integrates with the Mailboxer container correctly, ensuring that the intended output of sandbox matches the output of Mailboxer running locally - with there being three tests run per mailbox-annotated Erlang example. Deployment tests demonstrate portability of the containerised system across multiple environments such as: Windows, macOS, and Linux - which confirms the Dockerised architecture provides a consistent and lightweight runtime. A user study, is conducted via an online form, involving 5 (mixture of undergraduate/MsC/PhD) students and 6 industry professionals is conducted to assess the usability and effectiveness of MF, with additional feedback from the STARDUST research team which has been incorporated into MF. The study covered qualitative feedback and quantitative feedback, open-ended questions regarding their opinion, and tasks to perform. The results show that participants are able to understand key concepts (such as actor communication errors and mailbox types), navigate the website with ease, and successfully run examples in the sandbox. Quantitative results show high ratings, with 9.5/10 for navigability and 9/10 for clarity of sandbox output. Knowledge check questions are also answered correctly by majority of participants: 91% correctly identifying a payload mismatch, 100% correctly recognising when mailbox types were first introduced in literature. Qualitative feedback highlights MF as a clear and interactive way to learn about actor communication errors and behavioural typing. This evaluation showcases that the system functions as intended, but additionally meeting its goal of making advanced behavioural typing

more accessible and to make the underlying tool Mailboxer easier to use for a wider audience, with this evaluation in Chapter 5, Sections 5.1-5.4, and Appendix D.

Chapter 2: Literature Review

2.1 Data Types and Behavioural Typing

In Computer Science, software correctness (the assurance that the program behaves according to its specification) is a primary objective. Type systems prove to be very effective for improving the correctness of software.

Type systems, at their core, are a set of rules inside programming languages which classify their values and expressions, based upon their characteristics - these are known as *types*. Their purpose is to prevent undesirable behaviours (type errors) by ensuring that operations are performed on their compatible types e.g. `int + int = int`,
`int + string = type error`. This typically occurs at compile-time via static analysis, where the system examines the code and verifies that the types match i.e.
`1 (type Int) + 1 (type Int)  = 2 (type Int)`. If this check passes, then the type system provides a guarantee that these type-errors will not occur during execution. This is used to catch mistakes early in the development lifecycle [18].

Algol, being one of the most influential programming languages of the early era of Computer Science, introduced `if ... then ... else`, the `int n` declaration syntax, and the use of semicolons. Additionally, it popularized the term “type”. Within the report of Algol 58, it specifies the ”type” and ”class” of values on which variables are declared:

> Type declarations serve to declare certain variables, or functions, to represent quantities of a given class, such as the class of integers or class of Boolean values. [...] Throughout the program, the variables, or functions named by the identifiers I, are constrained to refer only to quantities of the type indicated by the declarator [19].

Since its introduction, it serves as the first checkpoint for a variety of programmatical errors that can occur. This has evolved into type-rules being put in place to dictate how different types of data are manipulated and combined. By understanding the various methods where type-checking can occur, it can further help understand why it is used (in the context of correctness of data manipulation and its limitations in concurrency), and how there is a need for more expressive behavioural type systems - such as mailbox types.

2.2 Static vs. Dynamic Typing

Type systems are divided into two categories: static and dynamic. In statically typed languages (such as Java, C++, and Rust), type-checking occurs during compile time. This allows errors to be detected early, and enables optimisations that can improve performance. In dynamically typed languages (such as JavaScript, Erlang, and Elixir), types are determined at runtime. This provides more flexibility, but comes at the cost of introducing runtime errors [20].

Static typing is associated with reliability and long-term maintainability. By verifying types before execution, the compiler can prevent different errors and generate optimised code. This makes static typing particularly useful for developers who are designing larger-scale projects which are intended to have a long lifespan - such as safety critical systems, where correctness and performance is a priority [21].

In contrast, dynamic typing allows variables to change their type during execution. This allows the code to be concise and expressive, which can allow developers to rapidly prototype projects and experiment easier. This is often used in the realm of web development, or early-stage projects (where speed to market matters more than reliability). The negative to this is that type-errors are only discovered during runtime, which leads to hidden bugs and degraded performance. As a result the responsibility for correctness is shifted from compiler to developer, who relies on comprehensive and extensive testing suites to catch these errors [20].

The choice between static and dynamic typing reflect the priorities of the project being developed: a financial trading platform picks a statically typed language to maximise safety and performance, meanwhile a startup (in the prototype stage) picks a dynamically typed language to iterate rapidly - at the cost of runtime errors.

2.3 Strong vs. Weak Typing

In addition to the differences between static and dynamic typing, programming languages are also split into being strongly or weakly typed. This distinction is independent of when type-checking occurs, and focuses on how strictly a languages enforced its type rules.

In a strongly typed programming language, operations on mismatched types are not permitted without explicit type conversion. This enforces clarity and intention of the code, which reduces subtle bugs, as the language itself will not perform implicit conversions. Erlang and Elixir (actor-based languages relevant to this project) are examples of a strongly typed language [21]. In comparison, a weakly typed language allows implicit conversion between types e.g. `String` to `Integer`. While convenient, it allows for non-obvious behaviour and bugs as a result - which are hard to trace to their point of origin [21].

This distinction can be useful, but it is also important to note that most modern languages are strongly typed, with weak typing being less common. For this reason, the remaining focus of this project will be on static and dynamic type systems, with attention to behavioural typing, as this more relevant to the challenges of an actor-based system.

2.4 Concurrency and the Limits of Data Typing

In a distributed and concurrent system, multiple threads (or processes) execute simultaneously via defined communications protocols. This raises the question: *Given the success of data types catching computational errors, could behavioural types help in catching communication errors?*

By extending these principles to communication patterns, it may be possible to prevent protocol violations and detect errors before they occur.

2.4.1 Shared-State Concurrency vs. Message Passing

In many languages (such as Java and C++) concurrency is implemented using a shared-state model, where multiple threads have access to the same memory, which can lead to complex issues such as: race conditions, dead-locks, and live-locks. These issues are difficult to reason about and debug [22], and are not type errors in a traditional sense - a type system can ensure that a bank balance is represented by a number, it cannot prevent two threads from simultaneously updating the memory that holds that number (which can result in corruption).

The actor model in Erlang is designed to avoid these problems. Each actor has exclusive access to its state, and communicates via asynchronous message passing. This eliminates the risks associated with shared-state concurrency, but also showcases the need for a type system that can reason about the behaviour of message exchanges, rather than only its structure.

2.4.2 Why Behavioural Typing Is Needed

These concurrency problems reveal a gap in what traditional type systems can do. They can verify what a value is, but not how its interactions should occur. Race conditions, dead-locks, and live-locks, are all examples of failure within the communication protocol - not data representation.

This motivates the development of behavioural type systems, which extend the guarantees of type systems from data to communication. By being able to describe and verify the sequence of interactions between concurrent components, behaviour typing allows for a method to provide static guarantees about the correctness of communication protocols. The actor model, which emphasises on message passing and isolation, provide a setting for exploring these ideas, and is the foundation for the introduction of mailbox types.

2.5 Behavioural Typing: Emergence and Development

Researchers are aiming to extend the success of data typing to detect communication errors with behavioural types. This research is aimed towards extending the power, and attention to detail, of type systems from the domain of data structures to communication protocols. The field of behaviour typing emerges as a result of this research, which is a paradigm that aims to specify and statically verify the communication behaviour of concurrent and distributed components.

This section defines core concepts of behavioural type theory, utilises the theory of session types (as an example to illustrate their mechanics), and analyse the limitations of this approach when applied to the actor model of concurrency.

2.5.1 Core Concepts of Behavioural Typing

Behavioural type systems are an extension of the traditional role that types in programming languages bring. Where data types describe the structure of data, and the valid operations that can occur, behavioural types describe the structure of communication and the valid sequences of interactions that a component does over time. The primary purpose of behavioural typing is to further enhance correctness and reliability of larger-scale, and more communication intensive systems - where concurrency problems are most common. This is done by providing a formal, and machine-checkable, language for describing communication protocols [2].

The scope in which behavioural typing can operate is broad, and covers concepts such as: component interfaces, communication contracts, and multi-party choreographies. By treating a communication protocol as a type, these systems can statically verify that a programs component follows a specified protocol. This allows for static detection from a range of critical errors which include:

- **Safety Properties:** The absence of communication errors (sending a message of an unexpected type) [2].
- **Liveness Properties:** The guarantee that a sent message will be received, or an interaction will terminate [2].

2.5.2 Session Types and Their Guarantees

The most mature and influential branch of behavioural type theory, introduced in early 2000s, is the theory of session types, which provides the realisation of the goals of behavioural typing, with the additional evolution of behavioural types illustrating their usefulness [23].

Session types are a typing discipline that describes a communication protocol as a sequence of interactions between communicating parties over a logical session. The core mechanism involves abstracting the entire protocol into a type: a simple client-server protocol can be specified as the client sends an `int`, receives a `String`, and then the session terminates. A session type system assigns a type representing this protocol to the communication channel connecting the client and server. The type checker verifies, statically, that both the client and server code implement their respective roles in the protocol correctly, which ensures the message of the right type is sent and received in the stated order.

It has been updated to handle more complex protocols, especially one with many participants such as:

Binary Session Types
Protocols with exactly two participants, based on duality: if one end sends a value of type T (`!T`), the other must receive it (`?T`). This ensures communication is always matched [24].

Multiparty Session Types (MPST)
Extends binary types to multiple participants. A global type specifies all interactions, which are projected into local types describing each participant's actions [25].

Just how data types evolved from simple to complex, the same applies to session types. This reflects how type systems must be more expressive to handle the increasing complexity, and interconnectedness, of systems being built [25].

Session type theory's main purpose, and why it is important, is called session fidelity (or protocol conformance). If all participants in the session pass their local type checks, the whole system is guaranteed to be free from certain communication errors. Specifically, it guarantees there wont be type mismatches in messages and importantly no communication dead-locks (where participants get stuck waiting for each other). This strong guarantee allows developers to check the overall correctness of a complex and distributed interaction by checking each individual part.

2.5.3 Limitations in Actor-Based Systems

Despite being powerful, session types are built upon a set of assumptions about the communication model. When applied directly to the actor model of concurrency, it reveals a mismatch. This mismatch is not a flaw in session type theory, but it shows that the actor model represents a different communication paradigm in which a different tool is required.

The key points for this divergence are: channel-centric vs actor-centric communication, unordered vs ordered communication.

Channel-centric vs Actor-centric Communication
Session types are channel-based: a session is a structured conversation over a dedicated channel, treated as a linear resource (used once, then discarded) [26]. In contrast, the actor model is address-based: actors communicate asynchronously via persistent mailboxes, which serve as their single point of contact [27].

Unordered vs Ordered Communication
Session types enforce a First-In, First-Out (FIFO) ordering of messages
(e.g., `!Int.?String.end` requires sending/receiving `Int` before `String`). However, actor mailboxes mix messages from multiple senders, so ordering is not guaranteed.

In Erlang, the `receive` construct allows for a selective receive, resulting in the actor being able to inspect and extract a message from it mailbox that matches a specific pattern - ignoring messages that may have arrived earlier. The ability to process messages, in any order, is a powerful feature of actor languages, but directly violates the sequential nature that is assumed of session types [5].

2.6 The Actor Model

The actor paradigm is important to understand the actor model. It is more than just a programming technique, but a comprehensive model of computation that helps us reason about concurrent operations. Originally defined in a 1973 paper, by Carl Hewitt, with further contributions from Peter Bishop and Richard Steiger. It drew influence from pioneering programming languages like Lisp and Simula, the object-oriented concepts of early Smalltalk, of capability-based security systems, and additionally the principles of packet-switched networks [28].

Hewitt's work is a response to the difficulties of managing concurrency with shared state, and this model proposes a shift in mentality with '*everything being an actor*'. This opposes the thought of '*everything is an object*', which is the main philosophy of object-oriented languages. The actor model simplifies building concurrent system by using actors as basic building blocks, which then handles the complex details of threads and synchronisation behind the scenes [29].

2.6.1 Core Principles

The model is defined by a small set of rules and components, which at their core, an actor is a computational entity that in the response of receiving a message and concurrently performing a set number of actions it specifically can:

- Send a limited number of messages to other actors, where their address is known.
- Create a limited number of new actors.
- Designate the behaviour to be used for processing the next message it receives.
  - This allows an actor to change its internal state over time.

These three actions are a complete and single step, and how these steps communicate depend on two other components.

Asynchronous Messages
Actors communicate by sending messages that cannot be changed, and the process is asynchronous. This means the sender does not block or wait for the receiver to process the message, and allows the advantage of parallel execution to occur [30].

Mailboxes
Each actor has its own private 'mailbox', which is a queue that stores incoming messages. When the message arrives, they are processed in a FIFO order. The mailbox acts as a buffer which can help with load management for the actor, resulting in communication flow being easier [30].

This design is most suitable for a distributed system, where all communication is done via passing messages. This process remains the same if the receiving actor is in the same process, different CPU core, or even on a separate machine across a network. This is known as location transparency, and is a cornerstone of the model's scalability and power [30].

2.7 Erlang and Concurrency

Originally developed at Ericsson in the mid 1980s to build highly available telecommunications systems in which its design choices reflect this [31], Erlang is not just a tool that supports actors. It is a complete system that was designed from the ground up around the principles of concurrency, distribution, and fault tolerance.

2.7.1 Processes as Actors

In Erlang, the item that corresponds to an actor is called a process. These are not heavy processes or threads like in an operating system, they are lightweight concurrency primitives managed by the Erlang Virtual Machine. A process can be created within microseconds and consume as little as 300 words of memory [32]. This efficiency makes Erlang particularly special, making possible for a single application to make thousands, or potentially millions, of concurrent processes. It allows the developer to adapt a process per task model, where each individual task (e.g. a web request) can be its own dedicated process. The granularity of concurrency allows isolation and parallelism that cannot be found with other language ecosystems [32].

2.7.2 The BEAM Virtual Machine

The secret sauce of Erlang is its virtual machine which is known as BEAM (Bogdan's Erlang Abstract Machine). BEAM is a complex runtime environment that is responsible for executing compiled Erlang code. In comparison to other languages that run on top of an operating system, BEAM has complex control over aspects such as process scheduling, memory management, and message passing [32].

It implements a pre-emptive scheduler, which ensures that one single process cannot take full ownership of a CPU core - to ensure responsiveness of the system. On a multi-core system, the BEAM generally has one scheduler thread per core, which has its own queue of processes. Additionally, it migrates processes between schedulers automatically, to balance load across cores. This deep integration into runtime is what allows Erlang to work concurrently, to help achieve its aims to be scalable, and to provide a safer and controlled environment for its approach towards fault tolerance [32].

2.8 Mailbox Types

The interaction between session types (ordered and channel-based) and actor communication (address-based and selectively received) introduce a number of challenges. Although session types have been successfully adapted to actor systems, such as Erlang [33], their channel-oriented foundations make them less naturally suited to the communication style of actors. To extend the safety guarantees of behavioural typing into the actor model, mailbox types have been developed as a behavioural type system designed specifically to reflect the asynchronous and selective nature of actor messaging. This section introduces mailbox types, explains their core mechanisms, and connects them directly to the Pat [5] and Mailboxer [4] tools which are subject of this project.

2.8.1 Actor Communication Model

To understand the design of mailbox types, it is important to understand the specific communication primitives of the actor model and what they are designed to capture. There are three key features of actor systems that distinguish them from channel-based models:

- **Asynchronous message passing:** Actors communicate by sending messages to a unique address of the receiving actor, and the actor does not block it instead continues its execution immediately [27].
- **Unified mailbox:** Each actor includes a single message queue, which is its mailbox. This receives messages from other messages in the system [27].
- **Selective receive:** An actor can look inside its mailbox and choose to process a specific message that matches a specific pattern, which can leave messages that arrives earlier for later processing [27].

This many-to-one unordered processing model of communication is what mailbox types are designed to formalise and verify.

2.8.2 Formalising Mailbox Contents

Mailbox types are a behavioural type system, first proposed in a process calculus setting by De'Liguoro and Padovani in 2018, and further developed for a full programming language by researchers at the University of Glasgow, as part of the STARDUST project [5] (https://epsrc-stardust.github.io/).

They represent a conceptual shift from session types. Instead of modelling a sequential dialogue, a mailbox type models the potential contents of an actors mailbox at any point in time. This innovation brought forward by these researchers enable the use of commutative regular expressions to describe the multi-set of messages that may be present in a mailbox - which neatly avoid the ordering assumptions of session types. The formal system, as described in the foundational papers, includes patterns and types [3] [5].

2.8.3 Pat and the Mailboxer Tool

The theoretical work on mailbox types has turned into a practical tool through the STARDUST (Session Types for Reliable Distributed Systems) research project. This project is a collaboration involving University of Glasgow, and is lead by key researchers in the field which include Professor Simon Gay and Professor Phil Trinder [34]. This project represents a shift with theory turning into an applicable tool.

The first significant outcome of this research done by STARDUST is Pat, which is a core programming language designed as the first implementation of mailbox types. As described in the ICFP 2023 paper, "Special Delivery: Programming with Mailbox Types", Pat provides formal semantics and an algorithmic type system that makes the theory of mailbox types checkable by a compiler, which address the practical language challenges like aliasing [5].

Building upon this foundation, the research team developed Mailboxer: a tool that bridges the gaps between the core language of Pat and a widely used language, Erlang. It functions as a translator tool that takes a subset of Erlang code, annotated by the developer with mailbox types, and proceeds to translate it into the Pat language. The Pat type checker then runs to statically verify that the Erlang code conforms to its specified communication protocols.

The progression of the theory of mailbox types, to the formulation of Pat, and further development of Mailboxer, demonstrates a path from initial idea and research into a final product that can be used by many. The work of this project, which is to generate a web front-end for Mailboxer, represents the next step in this progression. The goal is to make this static analysis technology more accessible, understandable, and usable for the community of Erlang developers - helping them further build more reliable systems. This reasoning provides justification for the project, allowing it to be a contribution towards the practical application of the key research done by the STARDUST team.

2.9 Supporting Tools and Architecture

There is now a firm understanding of the Actor model theory established, as well as its various implementations, outlined in previous sections. This section focuses on the development of the user-facing front-end using the React, and the strategy of packaging and deploying the application using Docker containerisation.

2.9.1 React in MF

React [12] is a widely used library within the JavaScript (JS) ecosystem, used for building intuitive and interactive user interfaces. It is chosen for this project for its component-based architecture, ecosystem, and suitability for creating a responsive single-page application (SPA). While React is designed for SPAs, this project requires a multi-page application (MPA) experience. This is done via `react-router-dom` [14], to simulate a multi-page navigation within an SPA. This allows for the user to navigate between sections without full-page reload, which improves the responsiveness and user experience.

Further details regarding implementation, including routing configuration and technical explanation of SPA vs MPA behaviour, are provided in Appendix A.1 and A.2.

2.9.2 Deployment and Portability

Modern applications often consist of multiple services (front-end, back-end, and additional services such as database etc.), which make deployment and consistency across environments a challenge. To address this, this project utilises Docker containerisation technology.

Docker [35] provides a lightweight and portable method to package applications with their core dependencies, to ensure the system behaves consistently across development, testing, and production environments. This approach eliminates environment-specific issues, such as *"it works on my machine"*. This allows for the MF front-end and back-end to be deployed reliably across different platforms.

Further details on Docker's benefits, and its core concepts, are provided in Appendix A.4.

2.10 Conclusion

The shift from traditional data typing to mailbox types represents a clear and logical progression for software correctness. While data types provide essential guarantees about integrity of data and its validity of operations, the introduction of concurrency reveals a limitation - they are unable to reason about the spacial and casual relationships that define interaction protocols. This introduces concurrency hazards such as: race conditions, dead-locks, live-locks, and synchronisation failures, which are not type errors but behavioural failures.

Behavioural type systems, such as session types, were developed to fill this gap by creating techniques to describe and verify communication protocols. The only issue is that the channel-based ordered nature of session types result in them not being an ideal fit for the address-based and unordered nature of communication in the actor model. This mismatch requires a need for a further 'type', being mailbox types.

Mailbox types model an actors mailbox as a commutative multi-set, to offer a system that naturally fit the actor paradigm. The research done by the STARDUST team, resulting in Pat, and Mailboxer, shows us that theory is able to be applied practically to provide static guarantees of protocol adherence and allow freedom from deadlocks for actor-based languages such as Erlang.

This project extends that work by creating a web front-end for Mailboxer, with the goals of making safety guarantees of mailbox types more accessible, and to help developers build more reliable systems with practical tools grounded in advanced programming language theory.

Chapter 3: Design of MF

3.1 Design Overview

The design to bridge the gap from theoretical foundations laid out by the STARDUST team (in their development of Mailboxer) to its practical implementation and integration into MF. Before development, the project required a concrete plan for the system architecture and user interface, This was done to ensure that the final product would be accessible, usable, and aligned with its main goals.

The design phase focused on two primary aspects:

- **System Architecture:** To define how the website front-end, server back-end, and Mailboxer would interact. This includes containerisation of the website front-end and back-end into one container, with Mailboxer into a separate container. This ensures a separation of concerns, allowing for portability and ease-of-deployment across multiple environments.
- **User Interface:** To plan the layout and navigation of MF, by utilising wireframes, to ensure that the website would have a clear (yet consistent) layout, and complex information (such as actor communication errors and behaviour types) are presented in a clear and structured manner.

The high-level design goals include:

- **Clarity:** To present complex concepts in an approachable and understandable way, to users unfamiliar with such concepts.
- **Accessibility:** To ensure that Mailboxer could be integrated with the website, without requiring local setup from the user.
- **Modularity:** To separate the user interface, server handling logic, and Mailboxer - to allow development and maintenance to occur independently of each other.

Wireframes (found in B.1) are created to visualise the intended user interface and navigation flow, for the end user. The system architecture is designed to balance simplicity with functionality. These design decisions provide a blueprint for the implementation of MF, which is described in Chapter 4.

3.2 System Architecture Design

The system is designed around a client-server-container model - with portability and maintainability as primary aims. It consists of two separate containers: one for the website (`mailboxer-website`), and one for Mailboxer (`paterl`). The website container combines the React-based front-end with the Express.js back-end server. The server is responsible for serving static front-end files, and to provide API endpoints for communication with Mailboxer. Mailboxer runs the research tool (`paterl`) in isolation, ensuring it can be invoked on demand without additional complexity within `mailboxer-website`.

This separation of containers is picked for several reasons:

- **Portability:** By containerising both Mailboxer and the website, they run separate and communicate via API endpoints (defined by the server). This system can be deployed on any system with Docker installed, allowing Mailboxer to be used without manual setup.
- **Maintainability:** `paterl` is kept as small as possible (under 2.5GB uncompressed - `slim` image tag), with `mailboxer-website` being under 350MB (uncompressed). This allows for additions to the website, or Mailboxer, to be done independent of each other - reducing overhead regarding maintenance.
- **Efficiency:** Once deployed, the container stack consumes no more than 80MB of RAM, making the system efficient to run on lightweight hardware.
- **Simplicity:** Early experiments of embedding `paterl` directly into `mailboxer-website` were rejected, as it introduced significant bloat to the container size, with the additional requirement of complex environment-specific dependencies. The final design avoids complexity by isolating `paterl` to its own container.

This interaction between services follows a clear flow. The user interacts with the front-end, which sends a request to the server via API endpoints, the server issues commands via `docker exec` to Mailboxer, with results being parsed and returned to the display on the front-end. This ensures a separation of concerns while remaining portable and deployable.

A simplified view of the architecture is shown below:

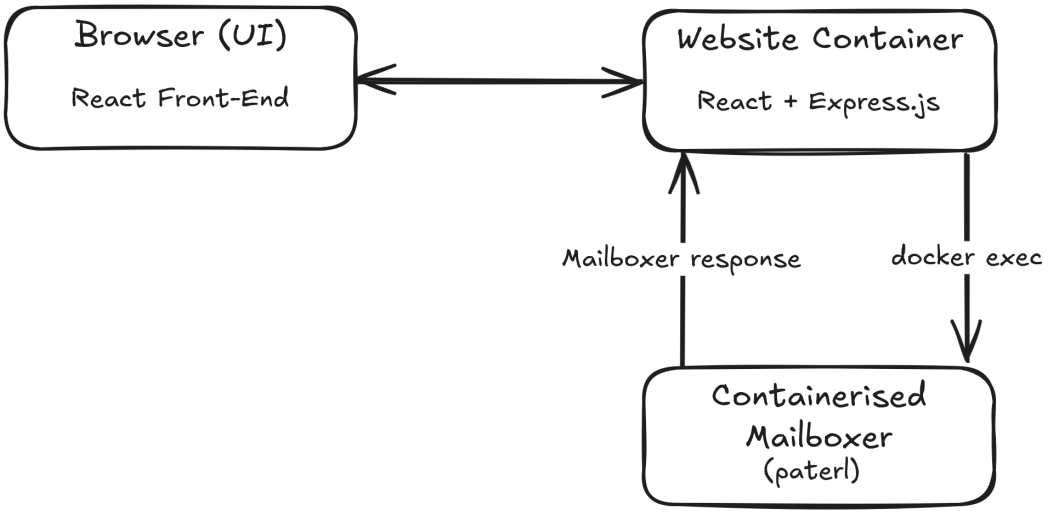

Figure 3.1: MF Software Architecture

3.3 User Interface (UI) Design

3.3.1 Wireframing

The wireframes (produced during the design phase) are to provide the initial blueprint for the overall layout of the website. They are made with the goal of ensuring navigational flow, page structure, and where information/features would live on certain pages. This allows the implementation phase to follow a clear structure, rather than adjusting the design in parallel with implementation.

Balsamiq [8] is used in the creation of the wireframes - to quickly develop low-fidelity designs, to prioritise structure and flow over visual detail. Each page is represented, which includes: Home page, Actor Communication Errors, Behavioural Types, Mailboxer introduction, Examples, Sandbox, and About. This ensures that the navigation between pages remains consistent, and information is presented in a clear and accessible manner.

The final implementation of the website has minor adjustments to the layout (compared to the wireframes), with the end results remaining inline with the original design. The wireframes act as the initial reference throughout implementation, ensuring the final website follows the intended design of layout and navigation. Emphasis was put on consistency and simplicity in the wireframes, which carries into the implementation phase - allowing abstract concepts (actor communication errors and behavioural types) to be accessible to developers.

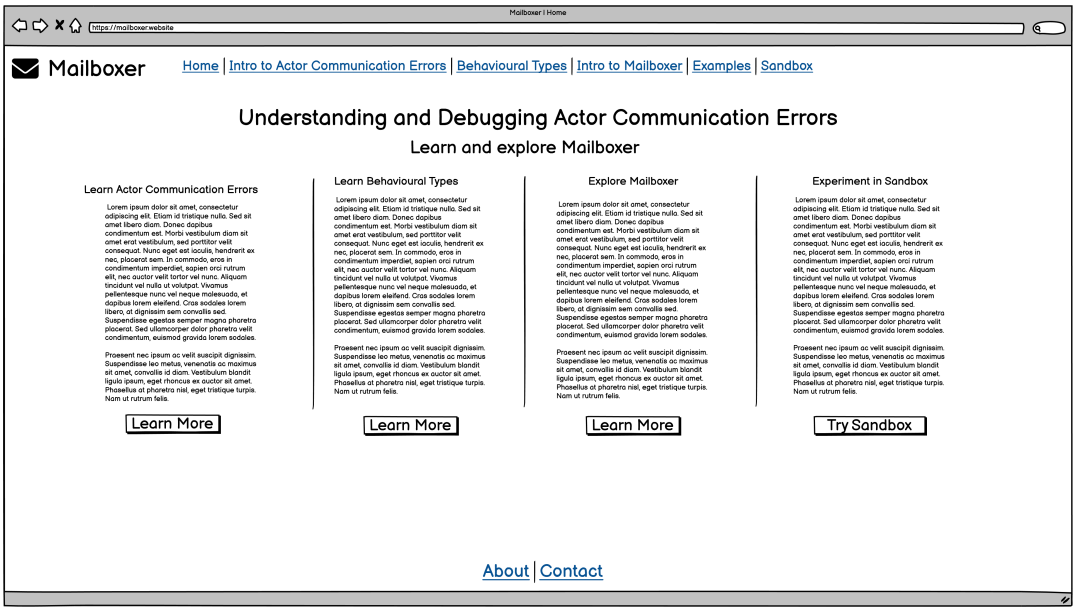

Figure 3.2: Wireframe of the Home Page.
The complete set of wireframes is included in Appendix B.1.

3.3.2 Page Layout and Structure

The structure of pages is designed to provided a clear and logical navigational flow - while keeping in mind that users of the website might not be familiar with the information contained in the website. The main goal is to group related concepts together, to present them in a consumable yet concise manner, and guide users throughout a progressing learning journey.

The navigation is designed to follow a primarily linear flow, with each page linking to the next, to form a 'story'. Concepts introduced in an earlier section are built upon (or utilised) in a later section, with the user being able to experiment with Mailboxer directly at the end. While there is a linear flow, the Home page provides links to specific sections, allowing users to skip directly to other sections.

This structure evolved during development, based on design iteration, and user feedback, to ensure clarity and accessibility for users. The final set of core pages in MF are as follows:

- **Home** – The entry point with an overview and navigation shortcuts to specific sections.
- **Actor Communication Errors** – Introduces common communication errors that can occur in actor-based systems, such as Erlang.
- **Behavioural Types** – Provides background on how behavioural types differ from traditional types, and how they prevent certain errors from occurring.
- **Mailboxer** – Explains the Mailboxer tool.
- **Examples** – Showcases Erlang code snippets with errors, and what Mailboxer would output on these specific errors.
- **Sandbox** – An interactive environment for running Mailboxer on provided several examples, code with errors, and a challenge example for users to edit.
- **About** – Provides project background, with links to the STARDUST project, related research papers, and source code for both the website and the Dockerised fork of Mailboxer.

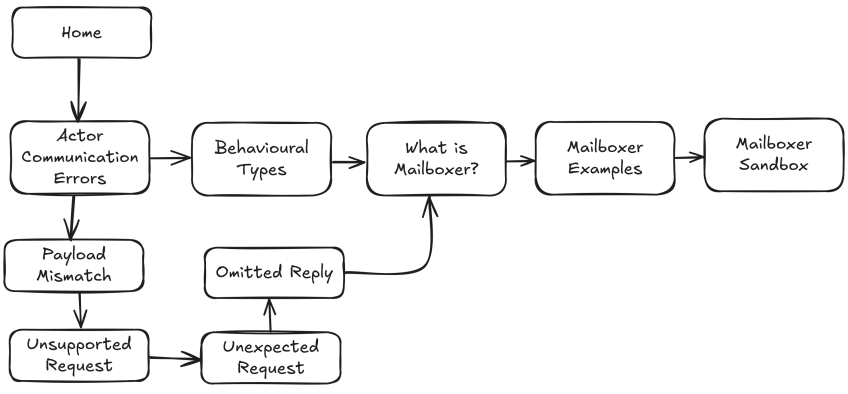

Figure 3.3: Website structure diagram of MF, rooted at the Home page.

This chosen structure mirrors a learning journey: beginning at theory, moving into the introduction of Mailboxer, and following up with utilising Mailboxer in real-time. This flow allows the user to learn the necessary context needed to understand the output of Mailboxer, and to utilise it effectively.

Consistency across pages is maintained through a shared navigation bar and footer, with a uniform layout containing consistent heading and text sizes, and syntax highlighting (where appropriate). This design allows for extensibility, with new pages being added by defining them as a new React component, and mapping it via `react-router-dom` - requiring minimal changes to the navigation bar, or the linking structure between pages.

3.4 Design Rationale

Several key design decisions are made in the website to ensure: consistency, accessibility, and safety for the intended audience. These decisions reflect the goal of making Mailboxer approachable to developers new to concepts explored in the website.

- **Routing**: Designed as an SPA, using `react-router-dom`, to simulate an MPA. This reduces perceived load time, after initial page load, allowing users to seamlessly navigate between pages (without full page refresh). This prioritises responsiveness and user experience of the website, over the simplicity of a traditional MPA.
- **Sandbox Scope:** The scope of the sandbox is intentionally limited to predefined examples (working and error), with an additional 'challenge' example. This is to keep the primary goal of the project inline with demonstrating the integration and capabilities of Mailboxer.
- **User Interface:** Bootstrap is used to prioritise consistency and accessibility of the website, to ensure a uniform layout (using consistent design patterns), with wireframes to guide content placement and navigation flow - to allow complex concepts to be presented in a simple and approachable manner.
- **Example Programs in the Sandbox:** The Sandbox contains a curated set of Erlang programs to demonstrate communication errors. These are as follows:
  - **Primary Working Example:**
    - `ID Server` – A program annotated with mailbox types, without errors.
  - **Error-Annotated Versions of ID Server:** Each illustrates a specific communication error:
    - `Unexpected Request`
    - `Omitted Reply`
    - `Payload Mismatch`
    - `Unsupported Request`
  - **Additional Working Examples:** More complex actor interactions:
    - `Parallel Fibonacci`
    - `Message Broadcast`
    - `Worker Pool`
    - `Ping Pong`
  - **Challenge Example:**
    - `Can You Fix This?` – combines multiple errors, encouraging users to apply their understanding to correct the program.

Overall, the design decisions ensure that the website remains lightweight, accessible, and provides the interactivity to demonstrate the capabilities of Mailboxer.

Chapter 4: Implementation of MF

4.1 Implementation Overview

The implementation of the project focuses on developing a web-based front-end to integrate Mailboxer, with the aim of making it accessible to developers (without the need of complex local setup).

The implementation of MF results in a 10-page website which is structured around detailed sections for explaining actor communication errors, behavioural types, and Mailboxer (a tool to transpile Erlang to Pat, a programming language design incorporating mailbox types). MF is implemented as a React-based single-page application (SPA), which which results in the full HyperText Markup Language (HTML) and JavaScript (JS), via the Express.js server, to be served to the user on initial load. This results in zero load time between pages, as the entire bundle is served to the browser. MF integrates directly with a Dockerised instance of Mailboxer. This allows the Mailboxer to be interacted with directly, within the Sandbox function of the website, with the server acting as a bridge between the front-end and Mailboxer.

The source code for the project is available at:
`https://github.com/ZubyWasTaken/mailboxer-website`

The implementation is containerised and comprises:

- A **Website container**, that combined the React front-end and Express.js server.
- A **Mailboxer container** which runs the tool Mailboxer, in isolation, where it can be integrated directly with the front-end via API endpoints defined in the server.

The front-end of MF is implemented using 6 libraries & tools (see Appendix C.2 for full details of front-end development stack), and the back-end is implemented using Express.js, with API endpoints exposed for communication between the frontend and Mailboxer (see Appendix C.1 for full list of API endpoints), with Docker serving as the technology using for containerisation.

4.2 System Architecture Implementation

The system is developed as a two-container stack, which consists of the website container and the Mailboxer container, which is referenced in Figure 3.1. The website container comprises both the React front-end with the Express.js server, while the Mailboxer container contains the compiled binaries needed for Mailboxer to operate. These two containers communicate via `docker exec` commands, that are issued by the server.

4.2.1 The Website Container

The website container is implemented in a `Node.js` environment and comprises:

- **React front-end:** Provides the user interface, served as static files.
- **Express.js back-end:** Exposes REST API endpoints to bridge communication between the front-end and the Mailboxer container.

These endpoints allow the website to load predefined examples, execution of Mailboxer commands, and to run Erlang code examples - which may have been edited by users. A full list of the API endpoints is provided in Appendix C.1.

4.2.2 The Mailboxer Container

The Mailboxer container is structured as a multi-language software stack:

- **Erlang/OTP:** Used to parse, preprocess, and lint Erlang source programs.
- **Pat Translation:** The Erlang syntax tree is translated into the intermediate Pat language, which is designed for communication safety.
- **Pat Type Checker:** Implemented in OCaml [36], as the `mbcheck` [37]. Utilises the Z3 theorem prover to solve constraints.

The resulting container contains the aforementioned software stack, which is then used to build and compile the binaries of Mailboxer. To utilise these binaries, the server executes commands within it on demand (without need of reconfiguration or rebuilding). The container is invoked via `docker exec` commands, in which Mailboxer is executed upon a file in question, that has been passed from the server to the container. This can be either predefined examples, or a temporary file which contains the user-written code generated from the Sandbox.

4.2.3 Deployment via Docker

The system is packaged using a multi-stage Docker build:

- *Build stage:* Compiles the React application using Vite and TypeScript.
- *Runtime stage:* Installs production dependencies, runs the Express server, and serves the compiled front-end.

This architecture allows for a separation of concerns: The React front-end provides the user with an accessible interface, the Express.js server manages the requests and interactions with Mailboxer, and Mailboxer performs the static analysis. This design ensures that the system is portable and deployable in different environments.

4.3 Front-End Implementation

4.3.1 Technology Stack

The front-end is implemented using a modern JavaScript (JS) & TypeScript (TS) stack, centred around React for its component-based model. In this model, the user interface is decomposed into independent, reusable components that can be used to build complex interfaces in a modular and maintainable way. Supporting tools are selected to provide fast builds, smooth navigation, responsive design, syntax highlighting, and the in-browser code editor.

This stack is chosen to balance development speed, performance, and ease of integration. A detailed breakdown of the specific libraries and their roles is provided in Appendix C.2.

4.3.2 Navigation and Page Structure

Navigation is handled by a central router in `App.tsx`, which maps URL paths to React components. The full routing configuration can be found in Appendix A.3.

4.3.3 The Sandbox Feature

The sandbox is the key component of the website. It provides an in-browser environment where users can experiment with Erlang code, and see the output of Mailboxer running on the code.

- **Code Editor:** Implemented using the `@monaco-editor/react` library, which allows users to view and edit Erlang code directly in the browser.
- **Output Panel:** Displays the results of Mailboxer analysis, including feedback regarding verification success and relevant output, or verification failure and relevant output.
- **Integration:** The editor communicates with the Express server, which forwards code to the running Mailboxer container via `docker exec`. The server then returns the processed output to the client.
- **Scope:** The Sandbox supports running predefined examples and user-modified code, but is not intended to be a full IDE. Its purpose is to provide an accessible way to explore communication errors and their detection by Mailboxer.

This feature allows the user to experience Mailboxer first-hand, and see how the output looks on a fully working code example, annotated with mailbox types versus several examples with errors included in them. It also gives the user an opportunity to solve errors contained within a 'challenge' by utilising knowledge learned in MF, and see the resulting output of Mailboxer as they try fix the example code.

4.4 Deployment of MF

The application is packaged and deployed using Docker to ensure consistency across development, testing, and production environments.

- **Multi-stage Docker build:**
  - *Build stage* – Compiles the React front-end using Vite.
  - *Runtime stage* – Runs the Express server and serves the compiled application.
- **Mailboxer container:** The Mailboxer analysis tool runs in a separate Docker container. The Express server communicates with this container via `docker exec` to perform static analysis.
- **Benefits:** Containerisation provides reproducibility, isolation, and portability. The same setup can be run locally or deployed to a remote server without modification.
- **Usage:** Running the application requires building the website container, or using a pre-build image, and ensuring the Mailboxer container is active. Once started, the system is accessible via a web browser on the exposed port.

This approach simplifies deployment, and ensures that the user environment is identical regardless of deployment platform - this reduces configuration issues and improves reliability.

Chapter 5: Evaluation of MF

The evaluation of the project is carried out by two complementary approaches. The first approach is done via functionality testing, with three tests run per mailbox-annotated Erlang example to verify that the system behaves as intended (10 examples total). The second being a user study, with 11 participants, to assess usability, clarity, and overall effectiveness of the website. The evaluation focused on the goals of the project, and to see if it met those goals.

Additionally, the STARTDUST team of researchers were contacted to provide feedback, with three researchers providing feedback: Professor Phil Trinder, Danielle Marshall (PhD), and Duncan Paul Attard (PhD). The resulting feedback, which was pertaining to layout, logo, and functionality, was adapted to be incorporated into MF, as to meet the goals of usability, clarity, and effectiveness.

5.1 Functionality Tests

The functionality of the system is verified through extensive manual testing, with three tests conducted per 10 of the predefined Erlang examples, with the aims of ensuring that all features and behaviour is as intended, and that the Mailboxer output in the website matches the expected output.

Testing covered navigation, predefined examples, the interactive Sandbox, error handling, and deployment across multiple environments. In all cases, the system behaved consistently and as intended, and the outputs of Sandbox matched those of Mailboxer running locally.

These results confirm that the website container communicates with the Mailboxer container, and that MF is portable across multiple platforms. A detailed breakdown of the testing process, and environments, is provided in Appendix D.1.

5.2 User Study Methodology

In addition to functionality testing, a user study is conducted. This is to evaluate the effectiveness of the website in teaching the core concepts needed to understand Mailbox types and Mailboxer, as well as the overall usability of the website. The aims of the study were to to determine if the website is able to present key ideas of actor communication errors, behavioural types, and information regarding Mailboxer, in a concise and accessible manner all while providing the user with a usable and easy-to-navigate interface.

Participants

A total of 11 participants consented to take part in the study. This group consisted of both students (Undergraduate/MSc/PhD) and industry professionals. This is done to provide a variety in background, and levels of programming experience, which allows the evaluation to capture different perspectives from individuals - ranging from those still in education, and those working in the industry.

Evaluation Procedure

Participants of the study were encouraged to explore the website, for no longer than 10 minutes, allowing them to freely interact with different sections of the website and the Sandbox. After 10 minutes, they were asked to complete an evaluation form, which is done via Google Forms. The full evaluation form can be found in Appendix D.2

The form consisted of three different categories of questions:

- **Background questions:** Role (student or developer) and years of programming experience.
- **Task-based knowledge checks:** Questions on actor communication errors, Mailboxer, and example outputs, which is designed to test if the website had effectively conveyed concepts to the user. Participants were allowed to refer back to the website while answering.
- **Usability feedback:** Likert-scale ratings [38] on navigation and clarity of Sandbox output, as well as open-text questions on what participants liked most and what they found confusing.

Analysis

Responses were analysed both quantitatively and qualitatively. Knowledge-check questions were marked against acceptable answers, which can be obtained directly from the website, and these results are presented in percentages of correct responses. Likert-scale ratings were averaged to provide an overall measure of usability, and open-text feedback is analysed to identify recurring strengths and areas of improvement.

Ethics

The initial section of the form outline the aims, purpose, the voluntary nature of participation, as well as their right to withdraw at any time. Email addresses were collected only for consent and withdrawal purposes, with participants being given clear instructions on how to withdraw their consent. No email addresses were utilised in the analysis or reporting of the results, and all results are anonymised in the final aggregation and presentation of data.

5.3 User Study Findings

A total of 11 participants completed the evaluation form, consisting of 5 students (mixture of undergraduate/MSc/PhD) and 6 industry professionals. Table 5.1 shows the distribution of participants by role and years of programming experience.

Role	0–1 yrs	2–3 yrs	4–5 yrs	6+ yrs	Total
Student	3	2	0	0	5
Developer	0	3	1	2	6
Total	**3**	**5**	**1**	**2**	**11**

Table 5.1: Participant breakdown by role and years of programming experience.

Knowledge-Check Questions
The knowledge-check questions were to test if participants understood key concepts presented in the website. The results were:

- Definition of a payload mismatch: 10/11 correct (91%).
- Impact of a payload mismatch: 9/11 correct (82%).
- Who introduced Mailbox Types (de'Liguoro and Padovani, 2018): 11/11 correct (100%).
- Example 4 error (Unsupported Request / Wrong message tag): 10/11 correct (91%).

These results indicate that the majority of participants were able to answer the questions correctly, which indicates that the website is effective in conveying the core concepts.

Sandbox
All 11 participants reported that they were able to successfully run all 5 working examples in the Sandbox, giving it a 100% success rate. This demonstrates the successful integration of Mailboxer with the website, functioning as intended for all users.

Usability Ratings
Participants rated the navigability of the website, and the clarity of the Sandbox output, on a 1-10 scale. The average scores were:

- Navigability: 9.5 / 10
- Clarity of Sandbox output: 9.0 / 10

These scores indicate that the website was perceived as easy to navigate, and clear in its presentation of results.

Open-Text Feedback

Analysis of the open-text responses revealed common strengths and areas for improvement:

- **Strengths:** Participants liked the simplicity and clarity of the layout, and the concise presentation of the information, as well as the inclusion of the interactive Sandbox feature to implement learned concepts as described in previous sections.
- **Areas for improvement:** Some participants noted that error highlighting in the Mailboxer examples could have been clearer, that the navigation could be further improved by addition of 'back' buttons in certain pages, and clearer tab indicators. There was also minor UI issues identified on smaller screens, such as the navigation bar alignment.

Overall, the user study results imply that the website is effective in teaching the core concepts of actor communication errors and core concepts of Mailboxer in a concise format, as well as providing a usable and intuitive interface for the users.

5.4 Performance Evaluation of MF

The website is tested for its responsiveness by measuring page load times on the deployed instance (`https://mailboxer.zuby.website`). Utilising Google Chrome [39], 10 refreshes of the home page were recorded, both with and without browser caching enabled - on a separate machine, not on the same local network as the website.

- **Without caching:** load times ranged between 436-500ms, with an average of 463ms.
- **With caching:** load times ranged between 216-403ms, with an average of 310ms.

This decrease in load time, with browser caching enabled, demonstrates that the single-page application design provides a fast and responsive user experience.

Execution of Mailboxer on all 10 example programs, in the sandbox is consistent, with results returning in less than 1 second, which is done manually, as the primary aim is to verify that the correctness of the sandbox is identical to the output of Mailboxer (compiled locally). This is measured by the response time of the API endpoint `/api/execute-paterl` (all API endpoints can be found in Appendix C.1), with the total Source Lines of Code (SLOC) of the combined examples that a user can run being just over 500, with the average example containing 50 SLOC. Additionally, Mailboxer runs on the same local network as the website container - so response time for this endpoint is kept minimal, as there is no external network latency associated with the communication of the containers.

For resource usage, the container stack uses a negligible amount of CPU usage (reporting as 0% on the Unraid [40] server), and between 50-80MB of RAM on average, never exceeding 80MB. This confirms that the system is both lightweight and efficient to run.

Portability
The Dockerised system is tested across multiple environments, with it being confirmed that MF behaves consistently with no difference in functionality or performance across platforms.

This includes:

- A local machine
- An M1 MacBook Air
- A local server, running Unraid 7.1.4

Full system specifications are given in Appendix D.1.5.

The usage of Docker ensured that the environment is reproducible and isolated, which required minimal setup beyond starting the container stack. Reverse proxying and external access via Cloudflare also worked without issue, which confirms that the system can be deployed in both local and production environments with relative ease.

Chapter 6: Conclusion

This project presents **MF**, a web-based front-end for Mailboxer, with the aim of making advanced concepts of behavioural typing and mailbox types accessible to a wider audience of developers and students. Mailboxer is a research tool that detects communication errors using mailbox types, but its accessibility is limited by the complex nature of its environment-specific setup and its command-line interface. MF addresses this issue by providing a browser-based platform that combines theoretical explanations, predefined mailbox-annotated Erlang examples, and an interactive sandbox - without requiring local setup.

6.1 Summary

The work undertaken in this project makes the following contributions:

1. **Background Research:** A review is conducted on type systems, behavioural typing, the actor model, and mailbox types. This establishes the theoretical motivation for mailbox types and grounded the development of MF in relevant academic work.

2. **Design of MF:** A learning-journey style website is designed, with wireframes to plan navigation, page structure, and presentation of abstract concepts. The design focused on accessibility and clarity, supporting a transition from theory to practice.

3. **Implementation of MF:** MF is implemented as a React-based single-page application, comprising approximately **3,000 source lines of code** written in TypeScript & JavaScript, Erlang, and Docker. The sandbox integrates directly with a Dockerised instance of Mailboxer and includes 10 predefined Erlang examples, annotated with mailbox types, (5 working, 4 error-annotated, and 1 challenge). The two-container architecture (website + Mailboxer) ensures portability, reproducibility, and isolation across environments. Once deployed, the system consumes only **50-80MB of RAM** and negligible CPU resources, making it lightweight and efficient to run.

4. **Evaluation of MF:** *Functionality testing* is carried out with 3 tests run per 10 of the predefined Erlang examples, confirming that sandbox results were identical to locally compiled Mailboxer outputs. Performance evaluation demonstrated responsiveness, with average remote home page load times of **463ms** (uncached) and **310ms** (cached), and sandboxed Mailboxer type checking analysis completing in under **1 second**, which is measured against the response-time of the `/api/execute-paterl` API endpoint. Portability is validated across **Windows 11**, **macOS (M1)**, and **Linux (Unraid)**.

   A *user study* with **11 participants** (5 students and 6 industry professionals) provided both qualitative and quantitative feedback. Knowledge-check questions showed that **91%** of participants correctly identified a payload mismatch, **82%** understood its impact, and **100%** correctly recalled when mailbox types were first introduced. All 11 participants were able to run the predefined sandbox examples successfully. Usability ratings were positive, averaging **9.5/10** for navigability and **9/10** for clarity of sandbox output. Feedback consistently highlighted MF's clarity, simplicity, and effectiveness as a teaching and experimentation tool.

6.2 Limitations

While usable and performant, MF has the following main limitations:

- **Restricted sandbox scope:** The sandbox is limited to predefined Erlang examples rather than providing a full integrated development environment (IDE) where users can upload and test their own programs.
- **UI constraints:** Minor interface issues were reported on smaller screens, such as navigation alignment and limited visibility of error highlighting in certain pages.
- **Scale of user study:** The evaluation involved only 11 participants. A larger and more diverse participant pool and a more challenging task list would strengthen the findings.

6.3 Future Work

There are several directions that future work of MF can include:

- **Extended sandbox functionality:** Support user-uploaded Erlang programs in addition to predefined examples, enabling further experimentation with mailbox types on users own code.
- **Enhanced error visualisation:** Improve error highlighting and introduce step-by-step debugging to make Mailboxer's verbose output more interpretable.
- **Integration with teaching tools:** Develop auto-marking tools to work with an expanded sandbox and allow for more interactive exercises, making it an educational tool for both undergraduate and postgraduate levels.
- **Scalability and industrial evaluation:** Conduct stress tests with larger codebases and more complex Erlang systems to assess MF's robustness and real-world applicability.
- **User study expansion:** Run a larger, more diverse user study with a more challenging task list to evaluate MF's teaching effectiveness.

Appendix A: Additional Background

A.1 SPA vs. MPA in React

Single-paged Application (SPA)

React [12] is a popular and powerful library in the JavaScript (JS) ecosystem for building intuitive and interactive user interfaces, albeit originally designed for single-paged applications (SPA). In a traditional multi-page website, the action of a user clicking on a link triggers a request to the server, which then returns a new HTML document, resulting in a full page reload.

In an SPA, the browser loads a single HTML file on initial load with the bundled JS code. From this point onwards, as the user interacts with the application and navigates between different pages and views, React intercepts these actions. Instead of requesting a new HTML document, React dynamically manipulates the Document Object Model (DOM) to change and rewrite the page contents. This gives the illusion that the user is navigating between different pages, without full page reload, resulting in a smoother experience [41]. To build a multi-page application, this behaviour must be simulated within the SPA constraints of React.

Multi-paged Application (MPA) with react-router-dom

The most common approach to implementing multi-page navigation in a React SPA is through `react-router-dom` [14]. This library provides the components needed to map URL paths to different React components, effectively creating distinct 'pages' within the application [42].

A more detailed description of the implementation steps is provided in Appendix A.2.

A.2 React Router Configuration

The following provides additional detail on how multi-page navigation is implemented in React using the `react-router-dom` library, with a the full `App.tsx` being showcased in Appendix A.3.

1. Installation: You add the library to your project using NPM.
2. Router Component: You wrap the root application component, typically `<App />` with a `<BrowserRouter>` component. This component uses the browser's History API to keep the UI in sync with the URL.
3. Route Definitions: Inside the application, you make use of the `<Routes>` component, which defines a collection of individual `<Route>` components. Each route maps a path (e.g., `/about`) to a specific element to be rendered (e.g., `<AboutPage>`).
4. Navigation Links: Use `<Link>` or `<NavLink>` components, which are provided by the library, instead of the standard `<a>` tag in HTML. These components prevent the default behaviour of the browser (full page reload) and instead update the URL programmatically, allowing the router to render the appropriate component.

A.3 Routing Implementation (App.tsx)

The following code shows the `App.tsx` file, which is utilised for the routing structure of the application using `react-router-dom`. This file maps URL paths to the corresponding React components, and allows for multi-page navigation of the website.

Listing A.1: App.tsx routing configuration

```
import { Route, Routes } from 'react-router-dom'
import Footer from './components/layout/Footer'
import Navbar from './components/layout/Navbar'
import About from './components/pages/About'
import ActorCommunicationErrors from './components/pages/
    ActorCommunicationErrors'
import BehaviouralTypes from './components/pages/BehaviouralTypes'
import Home from './components/pages/Home'
import Mailboxer from './components/pages/Mailboxer'
import MailboxerExamples from './components/pages/MailboxerExamples'
import OmittedReply from './components/pages/OmittedReply'
import PayloadMismatch from './components/pages/PayloadMismatch'
import Sandbox from './components/pages/Sandbox'
import UnexpectedRequest from './components/pages/UnexpectedRequest'
import UnsupportedRequest from './components/pages/UnsupportedRequest'

function App() {
  return (
    <div className="d-flex flex-column min-vh-100">
      <Navbar />
      <main className="flex-grow-1">
        <Routes>
          <Route path="/" element={<Home />} />
          <Route path="/actor-communication-errors" element={<
              ActorCommunicationErrors />} />
          <Route path="/actor-communication-errors/message-type-error/
              payload-mismatch" element={<PayloadMismatch />} />
          <Route path="/actor-communication-errors/message-type-error/
              unsupported-request" element={<UnsupportedRequest />} />
          <Route path="/actor-communication-errors/behavioural-type-error/
              unexpected-request" element={<UnexpectedRequest />} />
          <Route path="/actor-communication-errors/behavioural-type-error/
              omitted-reply" element={<OmittedReply />} />
          <Route path="/behavioural-types" element={<BehaviouralTypes />}
              />
          <Route path="/mailboxer" element={<Mailboxer />} />
          <Route path="/mailboxer-examples" element={<MailboxerExamples
              />} />
          <Route path="/sandbox" element={<Sandbox />} />
          <Route path="/about" element={<About />} />
        </Routes>
      </main>
      <Footer />
    </div>
  )
}

export default App
```

A.4 Docker: Concepts and Benefits

Why Docker?

Docker [35] is an open-source platform that automates the deployment of applications inside portable containers. A container is a package that contains the application code, and additionally the dependencies that it has (libraries, system tools, runtime versions etc.).

This approach solves the problem of '*well it works on my machine*' and provides benefits such as:

- **Consistency and Portability:** A container contains the exact environment required for operation. On execution, it operates identically - regardless of deployment platform. This eliminates bugs that can be caused by differences in the underlying host host environment infrastructure [43].
- **Isolation:** Each container runs its own isolated user-space on the host operating system kernel. This prevents communication between containers, or the host system, to improve security and stability. If a container crashes, or is compromises, then it is contained within the container and does not cascade onto other containers [43].
- **Resource Efficiency:** A container shares the host machines system kernel (rather than virtualising the entire hardware stack). This allows for it to be lightweight, and for several containers to run on a single host (compared to one host running virtual machines), this allows for a higher density of applications to run which is a more efficient usage of system resources [43].

Core Docker Concepts

The Docker workflow involves three key concepts: Dockerfile, Image, and Container:

- **Dockerfile:** A text file that contains instructions on how to build the image, which specifies the base image to start from (i.e. version of Node.js), and subsequent commands to copy application code, install dependencies, and commands to execute upon container creation [35].
- **Image:** The result of building a Dockefile. It is a lightweight read-only, stand-alone, and executable package which contains everything that an application requires to run. Typically stored in a registry (such as DockerHub), where they are shared among developers [35].
- **Container:** A runnable instance of the image. Upon execution, Docker creates a writeable layer on top of the read-only image, and the application code within is executed in isolation [35].

Appendix B: Design Artefacts

B.1 Mailboxer Front-End Wireframes

This section includes the wireframes created during the design phase of the website. They illustrate the layout, navigation flow, and intended user interactions. They are referenced in Chapter 3 (Implementation).

B.1.1 Home Page Wireframe

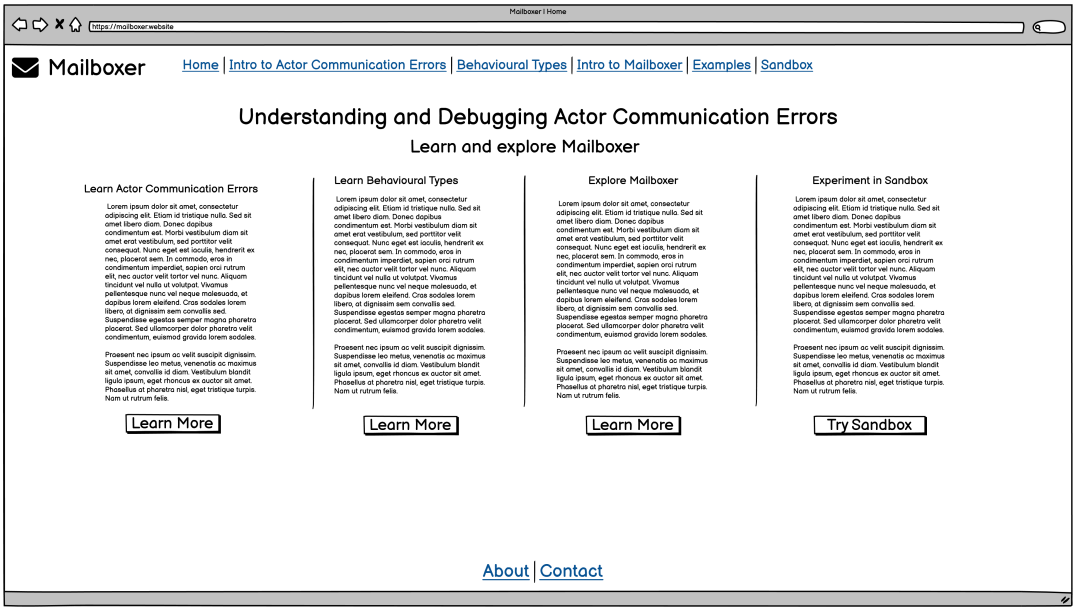

Figure B.1: Wireframe of the Home Page Wireframe

B.1.2 Actor Communication Errors Wireframe

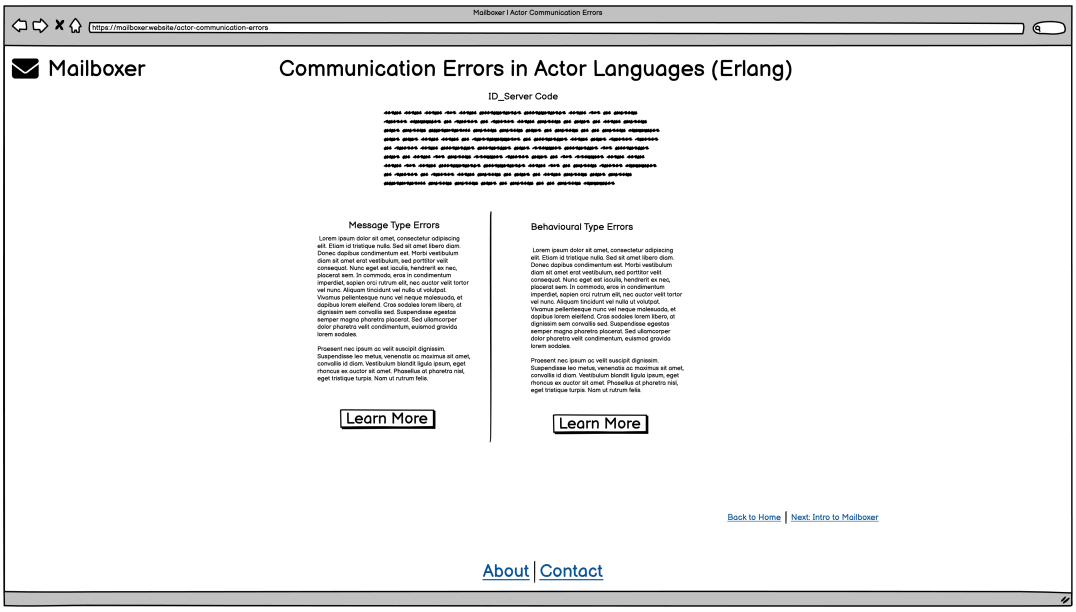

Figure B.2: Wireframe of the Actor Communication Errors Page

B.1.3 Message Type Errors Wireframe

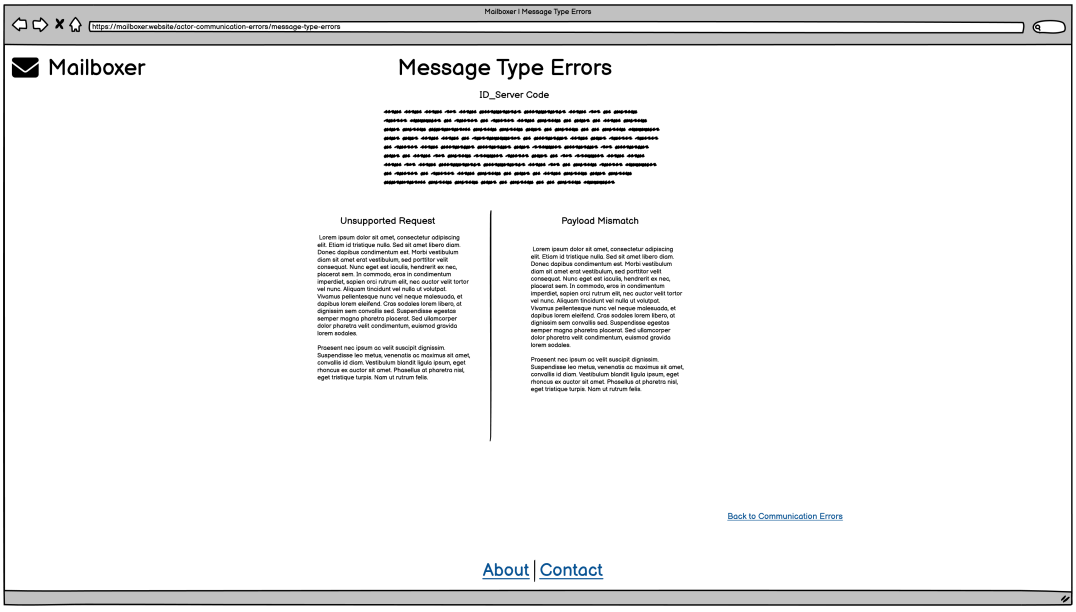

Figure B.3: Wireframe of the Message Type Errors Page

B.1.4 Behavioural Type Errors Wireframe

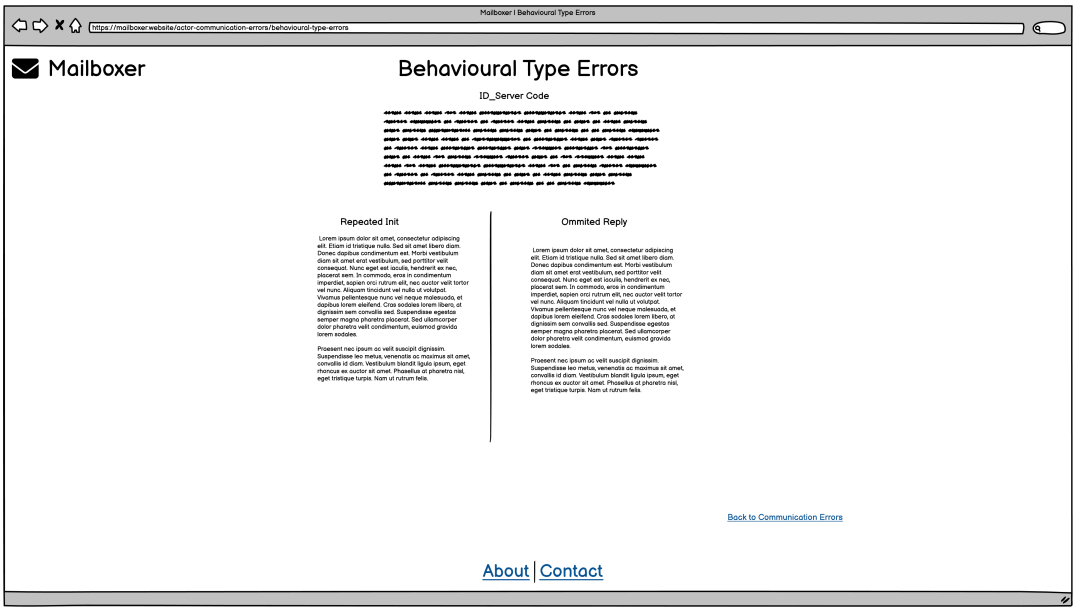

Figure B.4: Wireframe of the Behavioural Type Errors Page

B.1.5 Behavioural Types Overview Wireframe

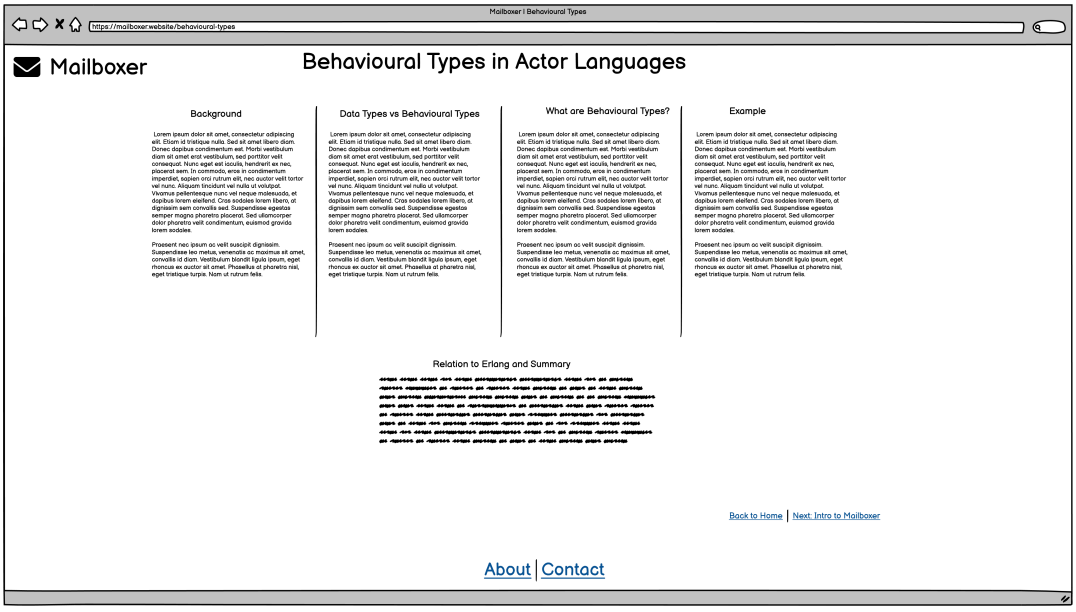

Figure B.5: Wireframe of the Behavioural Types Overview Page

B.1.6 Mailboxer Introduction Wireframe

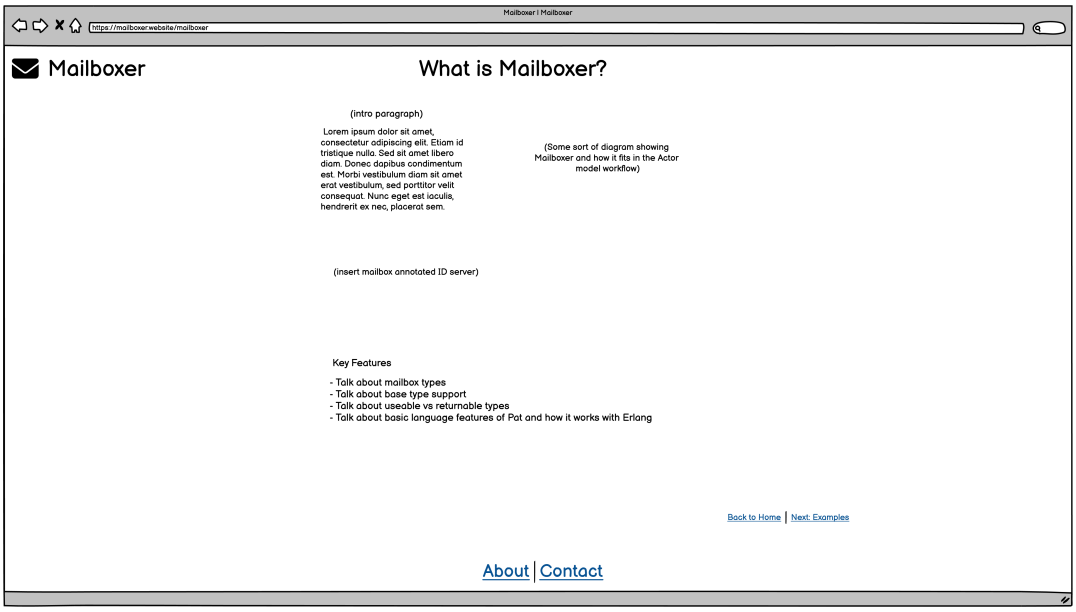

Figure B.6: Wireframe of the Mailboxer Introduction Page

B.1.7 Examples Page Wireframe

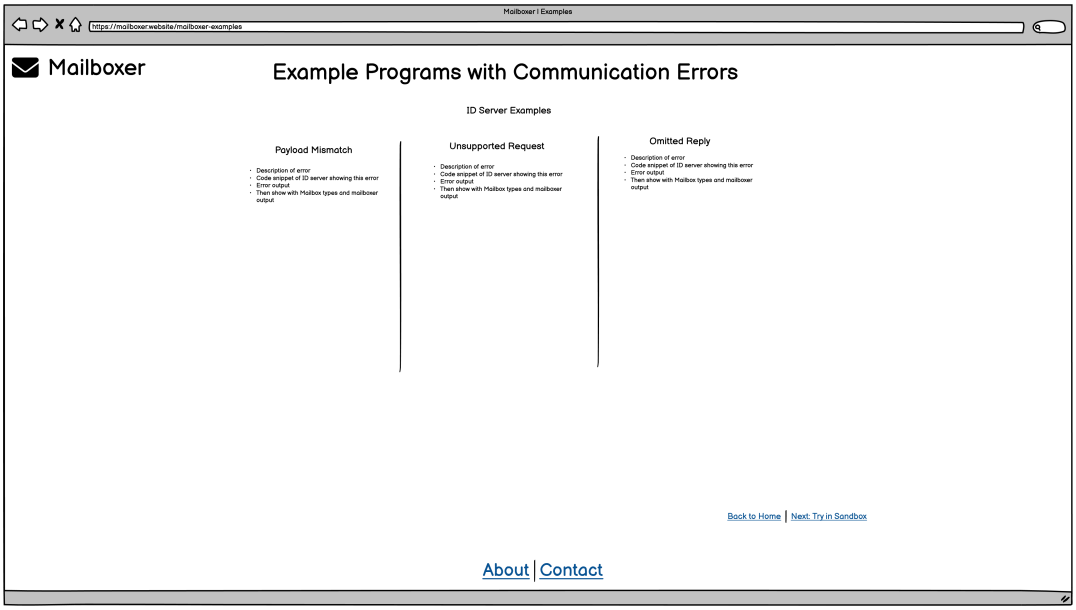

Figure B.7: Wireframe of the Examples Page

B.1.8 Sandbox Wireframe

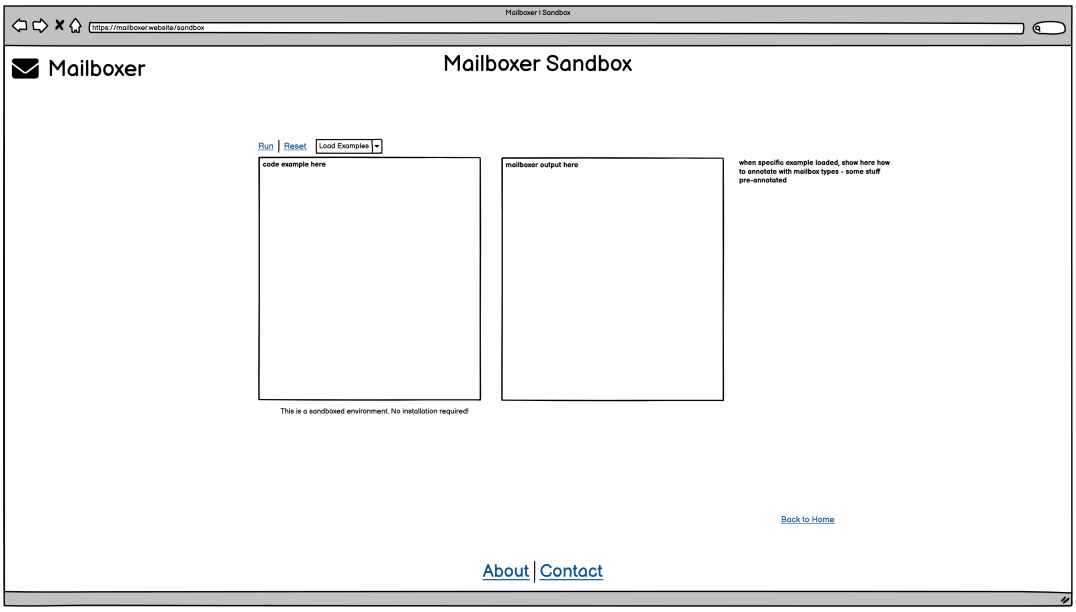

Figure B.8: Wireframe of the Sandbox Page

B.1.9 About Page Wireframe

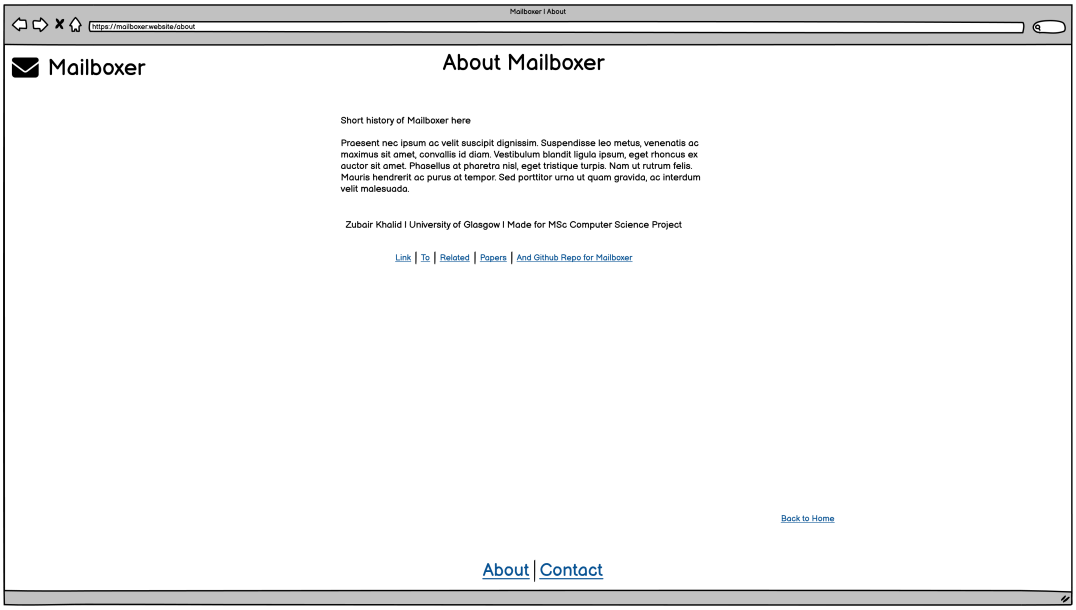

Figure B.9: Wireframe of the About Page

Appendix C: Implementation Details

C.1 API Endpoints of Website Container

The Express.js server in the website container exposes the following REST API endpoints to communicate with the Mailboxer container:

- `/api/read-example` – Loads Erlang example files from the `examples` directory.
- `/api/execute-paterl` – Executes predefined Mailboxer commands inside the Mailboxer container.
- `/api/run-code` – Runs user-provided Erlang code by writing it to a temporary file and executing Mailboxer on the file in the container.

C.2 Front-End Stack

The front-end is implemented using the following JavaScript/TypeScript stack:

- **React 19** – Chosen for its component-based model and strong ecosystem support.
- **Vite** – Lightweight build tool providing fast development server and production bundling.
- **react-router-dom** – Enables multi-page navigation within a single-page application.
- **Bootstrap / React-Bootstrap** – Used for responsive layout and consistent styling.
- **Highlight.js** – Provides syntax highlighting for code snippets, and within Monaco Editor.
- **@monaco-editor/react** – Provides the in-browser code editor for the Sandbox feature.

Appendix D: Evaluation Artefacts

D.1 Functionality Testing Details

D.1.1 Navigation Tests

All pages of the website were tested to confirm their contents, and navigation between pages is tested to verify that there is no full-page refreshes, and that URL slugs routed to the correct components.

D.1.2 Example Test Cases

Each predefined Erlang example is tested 3 times to confirm that they were served correctly via the server, and that the Mailboxer output displayed to the user is the expected output from the tool. To validate this, Mailboxer is setup manually following the recommended instructions (for WSL on Windows 11), and the identical commands that were executed in Sandbox were run manually on each example file. The outputs of the Sandbox and Mailboxer (local), were found to be identical.

D.1.3 Sandbox Tests

All examples were tested (10 total). The challenge example is modified with errors removed, to simulate the user solving these errors themselves. These examples were run through the Sandbox, and through Mailboxer (local), and the outputs matched exactly - which confirms that the website container correctly communicates with Mailboxer container, and provides accurate results.

D.1.4 Error Handling Tests

The system is tested under failure conditions, such as the Mailboxer container not running, or suddenly being forced offline. In this instance, the website would display an appropriate error message to the user, to ensure that the user is aware that Mailboxer is currently not operating - rather than failing silently.

D.1.5 Deployment Environment Testing

The container stack is tested across multiple environments to confirm portability. These included:

- A local machine, running Windows 11.
- An M1 MacBook Air, running macOS Sequoia 15.6.1.
- A local server, running Unraid [40] 7.1.4 - tested both on an internal IP and via an external domain (`https://mailboxer.zuby.website`) using Nginx Reverse Proxy Manager [44] and proxied behind Cloudflare [45] Domain Name System (DNS).

The system behaved consistently in all environments, which demonstrates the portability benefits of containerisation.

D.1.6 Testing Approach

All testing is done manually, as the primary aim is to verify that the output of the sandbox is identical to the output of Mailboxer (compiled locally). Automated testing is not utilised, as the focus is on validating on the correctness of the integration between the website container and Mailboxer.

D.2 Evaluation Questionnaire

MSc Computer Science Project Evaluation

This evaluation is being conducted by **Zubair Khalid (Student ID: 3062352k)**, MSc Computer Science, University of Glasgow.

The aim of this evaluation is to assess the usability and clarity of the Mailboxer website. You have been invited as a student or developer, as your feedback is valuable for evaluating the project.

You will be asked a few background questions, to complete short tasks and give feedback on the website. The results of this evaluation will be aggregated and anonymously displayed in the dissertation, and *only* the system is being evaluated.

Participation is voluntary, and you may withdraw at any time by contacting:
`3062352K@student.gla.ac.uk`

By continuing and entering your email address, you consent to take part.

Providing your email is only for consent purposes and to identify users who wish to withdraw their response. Your email will *not* be used in the final aggregation or display of the data.

Consent

- Email (required): _____

Background Questions

1. What best describes you? (required)
    - Student (Undergraduate/MSc/PhD)
    - Developer / Industry professional
2. How many years of programming experience do you have? (required)
    - 0–1 years
    - 2–3 years
    - 4–5 years
    - 6+ years

Actor Communication Errors

1. What is a *payload mismatch*? (required)
    - The server processes a request but fails to send the expected response.
    - The client sends a message with an incorrect or unrecognised tag.
    - The client sends a message with correct structure but wrong data type.
    - The client sends a valid message at the wrong time in the protocol.
2. What is one impact of a *payload mismatch*? (required) _____

What is Mailboxer?

1. In "What is Mailboxer?", there are Mailbox types defined on 3 separate lines. Please enter the **keyword** that defines a Mailbox type. _____

2. Who first introduced Mailbox Types, and in which year? (required) _____

Mailboxer Examples

1. In *Example 4*, what type of error is the code and Mailboxer output referring to? (required) _____

Mailboxer Sandbox

1. There are 5 *working examples*. Were you able to successfully run Mailboxer on these examples? (required)

   - Yes – all 5 examples worked
   - Yes – some of the 5 examples worked
   - No – none of the 5 examples worked

Usability Feedback

1. How would you rate the navigability of the website? (required)

   1 = very difficult | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 10 = very easy

2. When running the *Sandbox*, how clear was the output to you when running the examples? (required)

   1 = not clear at all | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 10 = very clear

3. What did you like most about the website? (required) _____

4. What did you find confusing about the website? (required) _____

Closing Note

Thank you for taking part.

The aim of this evaluation is to assess the usability and clarity of the Mailboxer website. Your feedback will help in the evaluation of the project.

If you have any further questions or comments, please feel free to contact:
`3062352K@student.gla.ac.uk`

Zubair Khalid (Student ID: 3062352k)
MSc Computer Science, University of Glasgow

References

[1] J. Armstrong and S. D. Pfalzer, *Programming Erlang: software for a concurrent world*, second edition ed., ser. Pragmatic programmers. Dallas, Texas ; Raleigh, North Carolina: The Pragmatic Bookshelf, 2013.

[2] D. Ancona, V. Bono, M. Bravetti, J. Campos, G. Castagna, P.-M. Deniélou, S. J. Gay, N. Gesbert, E. Giachino, R. Hu, E. B. Johnsen, F. Martins, V. Mascardi, F. Montesi, R. Neykova, N. N. Bono, L. Padovani, V. T. Vasconcelos, and N. Yoshida, "Behavioral Types in Programming Languages," *Foundations and Trends® in Programming Languages*, vol. 3, no. 2-3, pp. 95–230, 2016. [Online]. Available: http://www.nowpublishers.com/article/Details/PGL-031

[3] U. de'Liguoro and L. Padovani, "Mailbox Types for Unordered Interactions," *LIPIcs, Volume 109, ECOOP 2018*, vol. 109, pp. 15:1–15:28, 2018, artwork Size: 28 pages, 629960 bytes ISBN: 9783959770798 Medium: application/pdf Publisher: Schloss Dagstuhl – Leibniz-Zentrum für Informatik. [Online]. Available: https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.ECOOP.2018.15

[4] D. P. Attard, "paterl - Erlang to Pat transpiler," Apr. 2025, original-date: 2023-05-17T14:21:51Z. [Online]. Available: https://github.com/duncanatt/paterl

[5] S. Fowler, D. P. Attard, D. Marshall, S. J. Gay, and P. Trinder, "Special Delivery: Programming with Mailbox Types (Extended Version)," Jul. 2025, arXiv:2306.12935 [cs]. [Online]. Available: http://arxiv.org/abs/2306.12935

[6] Ericsson AB, "Erlang/OTP 28.0.2 — Erlang/OTP v28.0.2." [Online]. Available: https://www.erlang.org/doc/readme.html

[7] STARDUST, "Session Types for Reliable Distributed Systems." [Online]. Available: https://epsrc-stardust.github.io/

[8] Balsamiq, "About us." [Online]. Available: https://balsamiq.com/company/

[9] Microsoft, "Monaco Editor." [Online]. Available: https://microsoft.github.io/monaco-editor/

[10] "Express - Node.js web application framework." [Online]. Available: https://expressjs.com/

[11] Docker, Inc., "Dockerfile reference," 0200. [Online]. Available: https://docs.docker.com/reference/dockerfile/

[12] Meta Open Source, "React." [Online]. Available: https://react.dev/

[13] VoidZero Inc. & Vite Contributors, "Vite." [Online]. Available: https://vite.dev

[14] Shopify, Inc., "React Router Official Documentation." [Online]. Available: https://reactrouter.com/

[15] Bootstrap Team & Contributors, "Bootstrap." [Online]. Available: https://getbootstrap.com/

[16] "highlight.js." [Online]. Available: https://highlightjs.org/

[17] S. Atoyan, "suren-atoyan/monaco-react," Sep. 2025, original-date: 2019-06-16T07:46:31Z. [Online]. Available: https://github.com/suren-atoyan/monaco-react

[18] Tim Hurd, "Introduction to Data Types: Static, Dynamic, Strong & Weak — SitePoint," Jun. 2021. [Online]. Available: https://www.sitepoint.com/typing-versus-dynamic-typing/

[19] A. J. Perlis and K. Samelson, "Preliminary report: international algebraic language," *Communications of the ACM*, vol. 1, no. 12, pp. 8–22, Dec. 1958. [Online]. Available: https://dl.acm.org/doi/10.1145/377924.594925

[20] Kacper Rafalski, "Static vs. Dynamic Typing: Pros, Cons, and Key Differences," Oct. 2024, publication Title: netguru. [Online]. Available: https://www.netguru.com/blog/static-vs-dynamic-typing

[21] "Static vs Dynamic Typing: A Detailed Comparison," Aug. 2023, section: Technology. [Online]. Available: https://www.bairesdev.com/blog/static-vs-dynamic-typing/

[22] Beekey Cheung, "Solving Common Concurrency Problems," Dec. 2021. [Online]. Available: https://blog.professorbeekums.com/2021/solving-concurrency-problems/

[23] H. Hüttel, I. Lanese, V. T. Vasconcelos, L. Caires, M. Carbone, P.-M. Deniélou, D. Mostrous, L. Padovani, A. Ravara, E. Tuosto, H. T. Vieira, and G. Zavattaro, "Foundations of Session Types and Behavioural Contracts," *ACM Computing Surveys*, vol. 49, no. 1, pp. 1–36, Mar. 2017. [Online]. Available: https://dl.acm.org/doi/10.1145/2873052

[24] A. Scalas and N. Yoshida, "Less is more: multiparty session types revisited," *Proceedings of the ACM on Programming Languages*, vol. 3, no. POPL, pp. 1–29, Jan. 2019. [Online]. Available: https://dl.acm.org/doi/10.1145/3290343

[25] P.-M. Deniélou and N. Yoshida, "Multiparty Session Types Meet Communicating Automata," in *Programming Languages and Systems*, D. Hutchison, T. Kanade, J. Kittler, J. M. Kleinberg, F. Mattern, J. C. Mitchell, M. Naor, O. Nierstrasz, C. Pandu Rangan, B. Steffen, M. Sudan, D. Terzopoulos, D. Tygar, M. Y. Vardi, G. Weikum, and H. Seidl, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, vol. 7211, pp. 194–213, series Title: Lecture Notes in Computer Science. [Online]. Available: http://link.springer.com/10.1007/978-3-642-28869-2_10

[26] C. Sano, R. Kavanagh, and B. Pientka, "Mechanizing Session-Types using a Structural View: Enforcing Linearity without Linearity," *Proceedings of the ACM on Programming Languages*, vol. 7, no. OOPSLA2, pp. 374–399, Oct. 2023. [Online]. Available: https://dl.acm.org/doi/10.1145/3622810

[27] J. Masini and A. Francalanza, "Typing Actors using Behavioural Types," *Xjenza Online*, no. 1, pp. 51–55, Aug. 2015. [Online]. Available: https://doi.org/10.7423/XJENZA.2015.1.07

[28] C. Hewitt, P. Bishop, and R. Steiger, "A universal modular ACTOR formalism for artificial intelligence," in *Proceedings of the 3rd international joint conference on Artificial intelligence*, ser. IJCAI'73. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., Aug. 1973, pp. 235–245.

[29] "Wayback Machine," Sep. 2017. [Online]. Available: https://web.archive.org/web/20170924160220/http://www.dtic.mil/get-tr-doc/pdf?AD=ADA038246

[30] A. Stephanou, “Understanding the Actor Model,” Mar. 2025. [Online]. Available: https://mentorcruise.com/blog/understanding-the-actor-model/

[31] Thomas J. Bergin Jr., Richard G. Gibson Jr., “HOPL III: Proceedings of the third ACM SIGPLAN conference on History of programming languages,” in *HOPL III: Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages*. New York, NY, USA: Association for Computing Machinery, 2007.

[32] “The Hitchhiker’s Guide to Concurrency | Learn You Some Erlang for Great Good!” [Online]. Available: https://learnyousomeerlang.com/the-hitchhikers-guide-to-concurrency

[33] D. Mostrous and V. T. Vasconcelos, “Session Typing for a Featherweight Erlang,” in *Coordination Models and Languages*, W. De Meuter and G.-C. Roman, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, vol. 6721, pp. 95–109, series Title: Lecture Notes in Computer Science. [Online]. Available: http://link.springer.com/10.1007/978-3-642-21464-6_7

[34] “Session Types for Reliable Distributed Systems.”

[35] “What is a Container? | Docker.” [Online]. Available: https://www.docker.com/resources/what-container/

[36] “Welcome to a World of OCaml,” Aug. 2025. [Online]. Available: https://ocaml.org

[37] “SimonJF/mbcheck: Implementation of typechecker from paper ”Special Delivery: Programming with Mailbox Types”.” [Online]. Available: https://github.com/SimonJF/mbcheck/tree/main

[38] M. Koo and S.-W. Yang, “Likert-Type Scale,” *Encyclopedia*, vol. 5, no. 1, p. 18, Mar. 2025, publisher: Multidisciplinary Digital Publishing Institute. [Online]. Available: https://www.mdpi.com/2673-8392/5/1/18

[39] “Google Chrome – Download the fast, secure browser from Google.” [Online]. Available: https://www.google.com/intl/en_uk/chrome/

[40] “About Unraid.” [Online]. Available: https://unraid.net/about

[41] “Understanding Your UI as a Tree – React.” [Online]. Available: https://react.dev/learn/understanding-your-ui-as-a-tree

[42] “Routing.” [Online]. Available: https://reactrouter.com/start/framework/routing

[43] Hemanta Sundaray, “Docker Containerization: Key Benefits and Use Cases,” Feb. 2023. [Online]. Available: https://kodekloud.com/blog/docker-containerization/

[44] jc21.com, “Nginx Proxy Manager.” [Online]. Available: https://nginxproxymanager.com/

[45] Cloudflare, Inc., “Connect, protect, and build everywhere.” [Online]. Available: https://www.cloudflare.com/en-gb/