Ramsingh, Adrian (2023) *Improving interoperability in distributed multi-tier software stacks.* PhD thesis.

https://theses.gla.ac.uk/83372/

Copyright and moral rights for this work are retained by the author

A copy can be downloaded for personal non-commercial research or study, without prior permission or charge

This work cannot be reproduced or quoted extensively from without first obtaining permission in writing from the author

The content must not be changed in any way or sold commercially in any format or medium without the formal permission of the author

When referring to this work, full bibliographic details including the author, title, awarding institution and date of the thesis must be given

Enlighten: Theses
https://theses.gla.ac.uk/
research-enlighten@glasgow.ac.uk

Improving Interoperability in Distributed Multi-Tier Software Stacks

Adrian Ramsingh

Submitted in fulfilment of the requirements for the
Degree of Doctor of Philosophy

School of Computer Science
College of Science and Engineering
University of Glasgow

September 2022

Abstract

Distributed multi-tier software stacks organise and deploy software components as a hierarchy of interacting tiers. The components are typically heterogeneous, i.e. each component may be written in a different language and may interoperate using a variety of protocols. Tiered software is modular but leads to a range of interoperability challenges including the following. (1) Interoperating components in multiple languages and paradigms increases developer cognitive load since they must simultaneously reason in multiple languages and paradigms. (2) There must be correct interoperation of components, e.g. adherence to the API or communication protocols between components. (3) Interoperation between different components can lead to diverse modes of failure as each component can fail in unique ways. Many of these challenges are the result of contributing factors like tight coupling or polyglot programmming.

This thesis investigates techniques to improve heterogeneous interoperability in distributed multi-tier software stacks. Some common approaches include microservices and tierless languages.

Microservices are perceived to offer better reliability than components in multi-tier software stacks through the loose coupling of services. The reliability of microservices is investigated by combining the established properties of dependence and state with reliability. This defines a new three-dimensional space: the Microservices Dependency State Reliability (MDSR) classification with six classes. The feasibility of statically identifying MDSR classes is demonstrated with a prototype analyser that identifies all six classes in Flask microservices web applications. The reliability implications of the different MDSR classes are evaluated by running three case study applications (Hipster-Shop, JPyL & WordPress) against a fault injector. Key results are as follows. (1) All applications fail catastrophically if a critical microservice fails. (2) Applications survive the failure of individual minor microservice(s). (3) The failure of any chain of microservices in JPyL & Hipster is catastrophic. (4) Individual microservices do not necessarily have minor reliability implications.

In a tierless language, the compiler generates the code for each component and ensures their correct interoperation. They are mainly used to implement web stacks. However, their use in implementing IoT stacks is less common. This investigation compares interoperation in tiered and tierless IoT stacks through the systematic evaluation of four implementations of the prototype UoG smart campus IoT system: two tierless and two Python-based tiered. Key results

of the study are as follows. (1) Tierless languages have the potential to significantly reduce the development effort for IoT systems, requiring 70% less code than the tiered implementations. (2) Tierless languages have the potential to significantly improve the reliability of IoT systems. (3) The first comparison of a tierless codebase for resource-rich sensor nodes and one for resource-constrained sensor nodes shows that they have very similar functional structure and code sizes - within 7%.

Tier elimination is a technique that removes a tier/component by integrating two tiers. Specifically, this thesis investigates the implications of eliminating the Apache web server in a 4-tier web stack: Jupyter Notebook, Apache, Python, Linux (JAPyL) and replacing it with PHP libraries in the frontend webpage to get the 3-tier (JPL). The study reveals the following. (1) The JPL 3-tier web stack requires that the developer uses fewer programming languages and paradigms than JAPyL, i.e two compared with four languages and two compared with three paradigms. (2) JPL requires 42% less code than JAPyL. (3) In JPL, some of the functionalities can be automated due to the decreased abstraction levels at the upper layers of the stack. (4) However, the latency in JPL is two to three times greater than that of JAPyL. So while tier elimination reduces developer effort and semantic friction the tradeoffs are high performance overhead & resource consumption and increasing code complexity.

Acknowledgements

I would like to acknowledge my supervisors for their utmost support, dedication and patience during my PhD journey. My utmost thanks to Jeremy Singer for his enthusiasm, unique ideas and critical evaluations that assisted immensely in the shaping of this thesis. The weekly cricket conversations proved to be a delight. My sincerest gratitude to Phil Trinder for his unwavering support and belief in this thesis. His efforts in developing my writing skills and turning my engineering ideas into sound scientific research for this thesis cannot be underestimated. The work of these two, especially in handling someone with my day sleeping/night time awake hours, will become legendary.

Special thanks to Mart Lubbers and Pieter Koopman from the University of Nijmegen. Their collaboration and expertise on the BEST project was critical in the deepening of my research for this thesis.

Thanks to all my colleagues who worked in F101 over the years. I would like to acknowledge Dejice Jacobs, Cris Urlea, Thomas Kohler, Kyle Simpson, Yousef Alhaizaey and Sangkyu Park. The conversations and suggestions for my work over the years have proven to be very insightful and crucial in the development of this thesis.

I am especially grateful to my very good friend John Rae. His support over the years in Glasgow is worthy of mention. When we first met, I was struggling to adjust to life in Glasgow and becoming homesick. Now, not a week goes by that we are not visiting a pub or having some strange conversations about life and future events. Without him, pub nights and the loss of memories of the previous day would not have been possible.

Finally, I would like to acknowledge my parents, Claire and Yodhan Ramsingh, my aunts, Reena Nancoo and Pamela Dennis, and some very dear family friends especially Ganga and Dhana Singh for their unwavering support. Their tireless encouragement, motivation and constant social messaging have proven invaluable in being able to complete this journey.

This entire work is made possible by the Radboud-Glasgow Collaboration Fund, the UK EPSRC grants MaRIONet (EP/P006434) and STARDUST (EP/T014628). Thanks also to Lito Michala, Jose Cano, Greg Michaelson, Rinus Plasmeijer, and the many anonymous reviewers for valuable feedback on all published papers.

Contents

Acknowledgements

Declaration

1 Introduction **1**

1.1 Background . . . 1

1.1.1 Web & IoT Evolution . . . 1

1.1.2 Web Stacks . . . 1

1.1.3 IoT Stacks . . . 2

1.2 Interoperation in Multi-Tier Software Stacks . . . 3

1.3 Techniques for Improving Interoperation . . . 5

1.4 Research Questions & Contributions . . . 5

1.5 Publications & Authorship . . . 8

2 Background **9**

2.1 Conventional Web & IoT Stacks . . . 9

2.1.1 Conventional Web Tiers . . . 9

2.1.2 Notebook Programming . . . 10

2.1.3 JAPyL . . . 12

2.1.4 Conventional IoT Tiers . . . 13

2.1.5 Python Wemos Super Sensor (PWS) . . . 13

2.2 Comparing Web & IoT Stacks . . . 15

2.2.1 Differences between Web & IoT Stacks . . . 15

2.2.2 Similarities between Web & IoT Stacks . . . 15

2.3 Heterogeneous Interoperability in Distributed Multi-Tier Software Stacks . . . 16

2.3.1 Heterogeneous Interoperability . . . 16

2.3.2 Factors Affecting Heterogeneous Interoperability . . . 17

2.4 Techniques for Improving Interoperation . . . 21

2.4.1 Reducing the Number of Languages . . . 21

2.4.2 Microservices . . . 25

2.4.3 Tierless Languages . . . 27

2.5 Chapter Summary 30

3 Classifying the Reliability of Microservices 31
3.1 Motivation 32
3.2 Failure in Multi-Tier Software Stacks 33
3.2.1 Tight Coupling in Component Based Multi-Tier Software Stacks 33
3.2.2 Microservices Reliability 33
3.2.3 Partial Failures in Microservices 34
3.2.4 Identifying Causes & Detecting Failures in Microservices 35
3.3 Evaluating Failure in Component-Based Multi-Tier Software Stacks 35
3.3.1 Case Studies 35
3.3.2 Experiment Design 36
3.3.3 Catastrophic Failure in Component-Based JAPyL & WordPress 37
3.3.4 Failure in Microservices-Based JPyL 38
3.4 Classifying the Reliability of Microservices 39
3.5 Microservices Classification Case Studies 41
3.5.1 Hipster-Shop 41
3.5.2 JPyL ver2.0 42
3.5.3 WordPress ver2.0 43
3.6 Microservices Properties 43
3.6.1 Dependence 43
3.6.2 State 44
3.6.3 Reliability (Critical vs Minor) 45
3.6.4 Combining Microservices Properties 46
3.7 Catastrophic Failure in Microservices 46
3.7.1 Chained Failure 46
3.7.2 Patterns and Bad Smells 46
3.8 Microservices Dependency State Reliability (MDSR) Classification 48
3.8.1 Classification Schema 48
3.8.2 Semi-Automatic MDSR Classification 49
3.8.3 MDSR Patterns & Bad Smells Analysis 51
3.8.4 MDSR Pattern Instances & Implications 52
3.8.5 MDSR Bad Smells Instances & Implications 52
3.8.6 Semi-Automatic Pattern/Bad Smell Detection 53
3.9 Validating MDSR Reliability 54
3.9.1 Experiment Design 54
3.9.2 Critical Microservices Failure 54
3.9.3 Minor/Individual Microservice Failure 55
3.9.4 Critical/Chained Microservices Failure 55

3.9.5 Multiple Microservices Failure . . . 56
3.10 Evaluation Summary . . . 56
3.11 Chapter Summary . . . 58

4 Evaluating Tierless Languages for IoT Stacks 59
4.1 Detailed Authorship Statement . . . 60
4.2 Motivation . . . 60
4.2.1 Tiered IoT Stack Complexity . . . 60
4.2.2 Tierless Languages in IoT Software Stacks . . . 61
4.3 Smart Campus Case Study: Tiered Implementations . . . 62
4.3.1 Sensor Node Hardware . . . 62
4.3.2 PRS Implementation . . . 62
4.3.3 PWS Implementation . . . 63
4.3.4 Comparing PRS & PWS Implementations . . . 63
4.4 Understanding Tierless Clean iTasks/mTasks in IoT Programming . . . 67
4.4.1 Task Oriented Programming (TOP) . . . 67
4.4.2 Task Oriented Programming in Clean . . . 68
4.4.3 The iTasks Embedded DSL . . . 68
4.4.4 Engineering Tierless IoT Systems with iTask . . . 71
4.4.5 The mTask embedded DSL . . . 71
4.4.6 Engineering Tierless IoT Systems with mTasks . . . 73
4.5 Smart Campus Case Study: Tierless Implementations . . . 75
4.5.1 Tierless Implementations: CRS & CWS . . . 75
4.5.2 Operational Equivalence . . . 75
4.5.3 Functional Equivalence . . . 76
4.5.4 Memory Consumption . . . 77
4.5.5 Power Consumption . . . 77
4.6 Evaluation . . . 78
4.6.1 Is Tierless IoT Programming Easier Than Tiered? . . . 78
4.6.2 Comparing Codebases for Resource-Rich/Constrained Sensor Nodes . . . 80
4.6.3 Reduced Interoperation . . . 81
4.6.4 Automatic Communication . . . 82
4.6.5 High Level Abstractions . . . 83
4.6.6 Could Tierless IoT Programming Be More Reliable than Tiered? . . . 84
4.6.7 Maintainability . . . 86
4.6.8 Support . . . 87
4.6.9 Evaluation Summary & Reflection . . . 88
4.7 Chapter Summary . . . 88

5 Could Eliminating Tiers Improve Interoperation? 90
5.1 Motivation . . . 90
5.2 The Theory of Tier Elimination . . . 91
5.2.1 Component Modularity . . . 91
5.2.2 Tier Elimination . . . 91
5.3 Case Study . . . 92
5.3.1 Tier Elimination in JPL . . . 92
5.3.2 Implementing Server Requests in the Webpage . . . 93
5.3.3 Additional Functionalities . . . 94
5.3.4 Why PHP? . . . 95
5.4 Evaluation . . . 96
5.4.1 Experiment Design . . . 96
5.4.2 Programmability . . . 96
5.4.3 Abstraction . . . 97
5.4.4 Performance: Latency and Throughput . . . 98
5.4.5 Resource Usage: Core Utilization, Memory Overhead . . . 99
5.5 Evaluation Summary & Reflection . . . 100
5.6 Chapter Summary . . . 102

6 Conclusion 104
6.1 Summary . . . 104
6.2 Tradeoffs/Limitations . . . 107
6.2.1 Interoperability . . . 107
6.2.2 Microservices . . . 108
6.2.3 Tierless Languages . . . 109
6.2.4 Tier Elimination . . . 110
6.2.5 Research Limitations . . . 110
6.3 Future Outlook & Recommendations . . . 111
6.3.1 Microservices . . . 111
6.3.2 Tierless Languages . . . 112
6.3.3 Tier Elimination . . . 112
6.4 Future Work . . . 113

A Interoperability in JAPyL Web Stack 115
A.1 Interoperation between Apache & Jupyter Components . . . 115

B Declarative Programming in JAPyL 118
B.1 JAPyL Declarative Security Examples . . . 118
B.2 Developer Effort - Links vs PHP . . . 119

C Microservices in JPyL Web Stack **120**
C.1 JPyL Individual Port Config/Reverse Proxy Microservice 120
C.2 Reliability Experiment Code Fragments 120
C.3 JPyL Port Config/Reverse Proxy Microservices Chain 121

D PRS & PWS Sensor Code **123**
D.1 Light Sensor Code for PRS 123
D.2 Light Sensor Code for PWS 125
D.3 Comparing PRS & PWS Temperature Sensor Code 126

Bibliography **131**

Glossary **149**

List of Tables

2.1 Language Paradigm Classification 19
2.2 Examples of Tierless Languages based on Compilation Scheme 27
2.3 Examples of Tierless Languages based on Remote Communication Techniques 28

3.1 Microservices Classification Criteria 44
3.2 Microservices Patterns 47
3.3 Microservices Bad Smells 47
3.4 MDSR Pattern Classification. 48
3.5 MDSR Bad Smell Classification. 49
3.6 MDSR Classifications of 7 small Flask Applications containing 30 Microservices 52
3.7 MDSR Patterns/Bad Smells found in 7 small Flask Applications containing 30 Microservices 53

4.1 UoG Smart Campus Sensor Nodes: Maximum Memory Residency (Bytes). . . 77
4.2 Comparing tiered and tierless smart campus code sizes: SLOC and number of source files. PWS and CWS execute on resource-constrained sensor nodes, while PRS and CRS execute on resource-rich sensor nodes. 79
4.3 Smart Campus implementation languages and paradigm comparison. 81
4.4 Comparing Clean and Python Programming Abstractions Using the PWTS and CWTS Temperature Sensors (SLOC and Total Number of Files.) 84

5.1 Feature Similarities Between PHP & Python 95
5.2 Lines of Code (LOC) 96
5.3 Programming Language Implementation 97
5.4 Cyclomatic Complexity (cc) 97
5.5 JAPyL vs JPL Security Implementations 98

List of Figures

1.1 Web Stack Architecture 2
1.2 A 4-Tier LAMP Web Stack 2
1.3 PWS Distributed IoT Architecture 3
1.4 JAPyL Distributed Architecture 4

2.1 N-Tier Architecture 10
2.2 Jupyter Kernel Component 11
2.3 Jupyter Subkernel Integration 11
2.4 JAPyL Defense-in-Depth Security Model 12
2.5 A Generic 4-Tier IoT Architecture 14
2.6 PWS Distributed IoT Architecture 14
2.7 Example Comparison of Protocols in Web vs IoT Stacks 16
2.8 Inverse Pyramid Classification 20
2.9 Java Beans Architecture 22
2.10 4-Tier Single Language MEAN Architecture 23
2.11 Virtual Machine Code Translation Process 23
2.12 API Protocols in PWS Application Layer 24
2.13 Layered Microservices Model 26

3.1 A Chained Microservice Example 33
3.2 JAPyL 4-Tier Architecture 36
3.3 JPyL 4-Tier Architecture 37
3.4 Component Based WordPress Architecture 38
3.5 JAPyL Apache Failure at 43s. 38
3.6 WordPress Failure at 43s. 38
3.7 JAPyL vs JPyL Reliability Comparison 39
3.8 Hipster-Shop Architecture 41
3.9 JPyL with Different Classes of Microservices 42
3.10 WordPress Microservices Application 43
3.11 Stateful Microservices Example 45
3.12 MDSR Classification Tree. 48

3.13 JPyL Output From MDSR Analyser 50
3.14 IBM Worklogs Output From MDSR Analyser 51
3.15 JPyL PortConfig & ReverseProxy Critical Failure at 43s. 54
3.16 JPyL SSL Critical Failure at 43s. 54
3.17 WordPress HTTP REST & Comment Critical Failure at 43s. 55
3.18 Hipster Cart & Payment Services Critical Failure at 43s. 55
3.19 JPyL Minor/Individual Failure (Service Logging) at 43s. 56
3.20 JPyL Chained (UserData & White/Blacklisting) at 43s. 57
3.21 Hipster Chained (Product & Recommended) at 43s. 57
3.22 JPyL Multiple Minor Failures at 16s, 32s, 48s. 57

4.1 Exposed views of sensor nodes: the Wemos on the left is used in PWS and CWS; the Raspberry Pi on the right is used in PRS and CRS. 62
4.2 iTask SimpleTempSensor: Webpage (left) and Deployment Diagram (right). . . 69
4.3 Web pages generated by the TempHistory tierless web application: on the left sorted by time; on the right sorted by temperature. The Take button is only enabled when the topmost editor contains a positive number. 71
4.4 Deployment diagram of the iTask TempHistory tierless web application from Listing 4.7. 71
4.5 Tierless iTask CRTS temperature sensing IoT system: web page (left) and deployment diagram (right). 73
4.6 Tierless iTask/mTask CWTS temperature sensing IoT system: web page (left) and deployment diagram (right). 73
4.7 PRS and PWS (left) together with CRS and CWS (right) mapped to the 4-tier IoT architecture. Every box is the diagram denotes a source file or base. Bold blue text describes the language or technology used in that source. The Network and Perception Layers are unique to the specific implementation, where the Application and Presentation Layers are shared between implementations. . 75
4.8 Web interfaces for CWS and CRS (left); PWS and PRS (right). 76
4.9 Comparing the percentage of code required to implement each functionality in tiered/tierless and resource-rich/constrained smart campus implementations. . . 79

5.1 n-Tier Architecture Variation 92
5.2 3-Tier JPL Architecture 93
5.3 JAPyL vs JPL Latency (RaspberryPi) 99
5.4 JAPyL vs JPL Latency (Docker) 100
5.5 JAPyL vs JPL Throughput (RaspberryPi) 101
5.6 JAPyL vs JPL Throughput (Docker) 102
5.7 JAPyL vs JPL CPU Resource Utilisation (RaspberryPi) 103

Declaration

The work reported in this dissertation is primarily my own with the exception of Chapter 4 that is collaborative work [87]. The original University of Glasgow smart campus implementation (PRS) was implemented by Jeremy Singer, Dejice Jacob and Kristian Hentschel using Python on the RaspberryPi [59]. The tierless IoT stacks of CRS and CWS were implemented by Mart Lubbers and Pieter Koopman from the University of Nijmegen.

Chapter 1

Introduction

1.1 Background

1.1.1 Web & IoT Evolution

In less than thirty years, the World Wide Web (WWW) has undergone a rapid evolution. Simple static web apps developed using languages like XHTML, CSS & JavaScript have now become more dynamic. Here, data driven apps like social media portals now utilise languages like PHP, Python, Perl or Java alongside data stores like MySQL, MongoDB or Redis [103].

Now another evolution is taking place where any physical device like coffee makers, washing machines or watches are connecting to the internet and communicating with one another. There is now an interaction between the physical and digital worlds commonly referred to as the Internet of Things (IoT) [41].

Traditionally, applications in both of these domains have been developed and implemented as tiered distributed software. They comprise several layers of interoperating software components on different nodes [20].

1.1.2 Web Stacks

Web stacks combine software components to implement a web application. A component may be defined as an independent, high performance, multi-threaded unit of software with a high level of abstraction, designed to perform a single functionality [30].

At a minimum, a web stack consists of an operating system, data store, web server and a client side interface like a web browser shown in Figure 1.1. Early web stacks combined Linux, Apache, MySQL & PHP (LAMP) shown in Figure 1.2. In 2020, implementation of this 4-tier stack still remains common as 60% of all WordPress applications hosted online utilise LAMP [64]. However, more modern web stacks have evolved. Examples include MongoDB, ExpressJS, AngularJS & NodeJS (MEAN) and Jupyter, Apache, Python & Linux (JAPyL) discussed in Section 2.1.

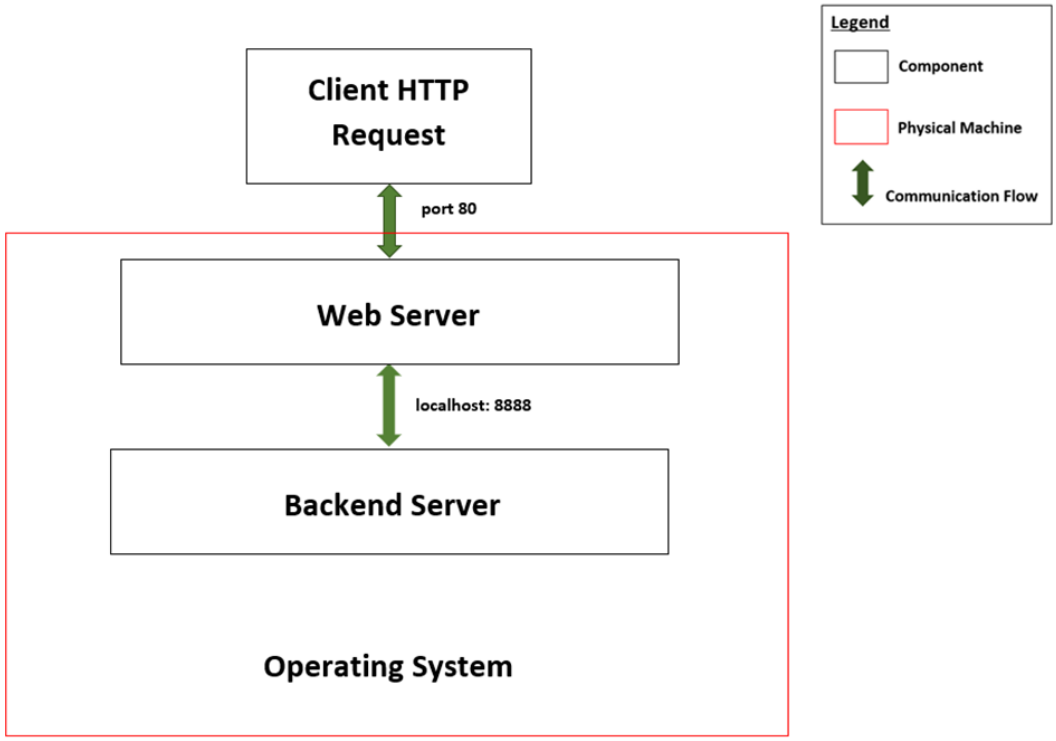

Figure 1.1: Web Stack Architecture

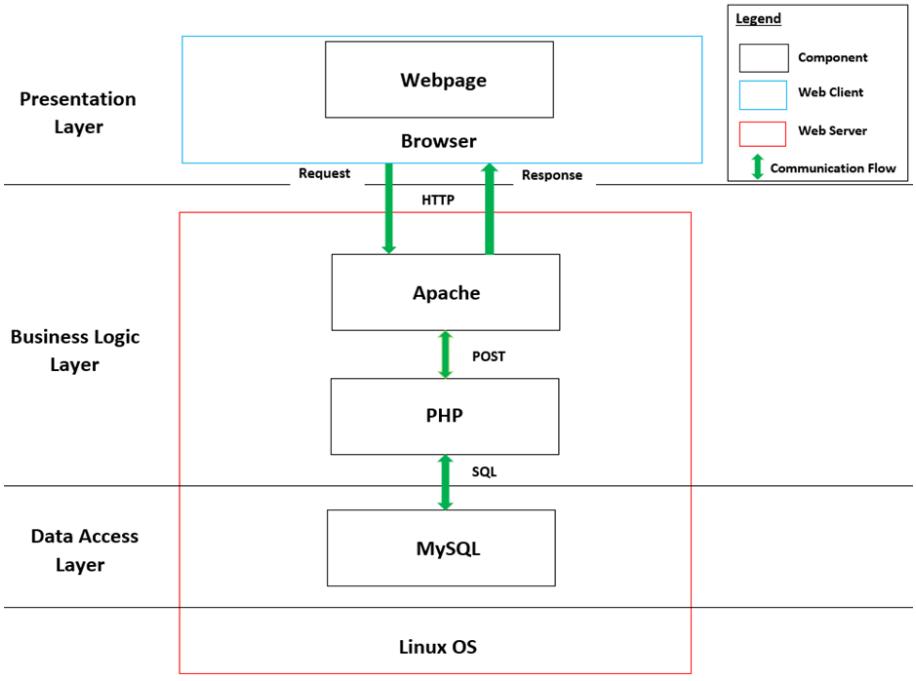

Figure 1.2: A 4-Tier LAMP Web Stack

1.1.3 IoT Stacks

IoT stacks are even more complex than web stacks. They are not just software based as there is also interaction with the physical world. They combine a web application with hardware like

embedded sensors and actuators. This allows the application to collect and aggregate data [87]. One example is the (Micro)Python, Wemos System (PWS) stack shown in Figure 1.3 and discussed in Sections 2.1.5 & 2.4.

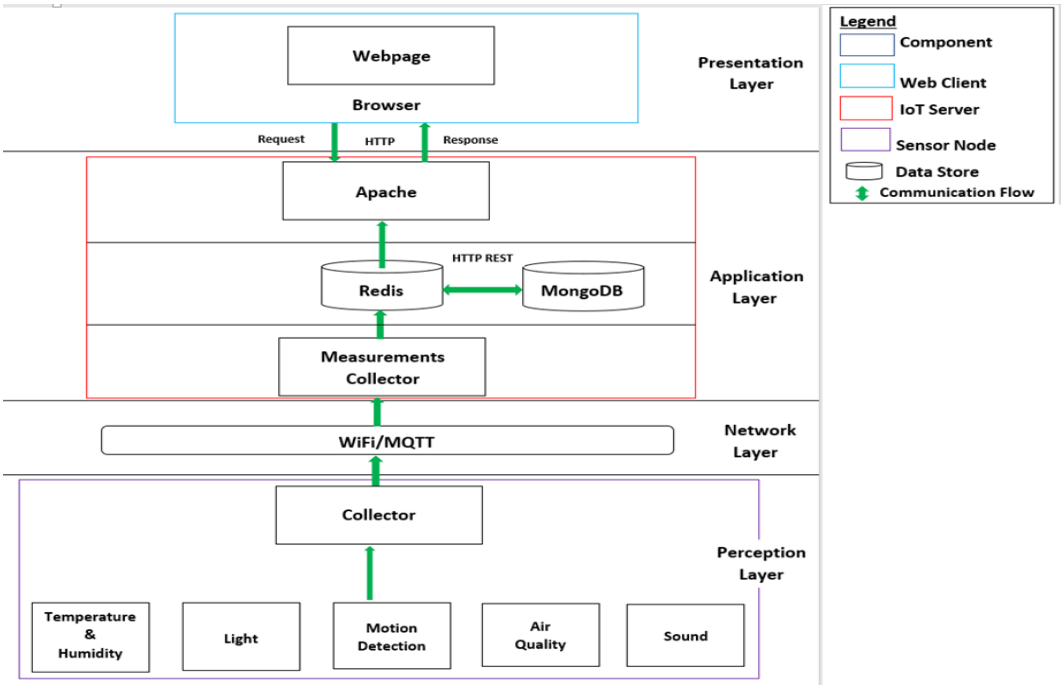

Figure 1.3: PWS Distributed IoT Architecture

1.2 Interoperation in Multi-Tier Software Stacks

A feature of most multi-tier software stacks is heterogeneity. There are multiple different components, programming languages and protocols that must interoperate in the stack to achieve the business function of the application [100]. One example is the JAPyL web stack shown in Figure 1.4.

The purpose of this stack is to make Jupyter Notebooks web accessible. The Notebook component is usually embedded into a webpage with an Apache Web Server as a reverse proxy. Here, Apache is usually interposed between the client and the Jupyter Server, taking requests from clients and forwarding them to Jupyter using the HTTP protocol [99]. Appendix A.1 provides an example of the sorts of configurations a developer has to implement in both Apache and the Jupyter Server to allow for communication between the two components.

This creates heterogeneous interoperability as different forms of interoperation like syntactic and semantic interoperability can occur [5]. Here different components are able to (1) read and interpret the format of messages being exchanged (2) understand the meaning of data in a similar way. For example, in the LAMP web stack, the PHP & MySQL components are able to interoperate because PHP passes messages to the MySQL component as SQL messages. This is

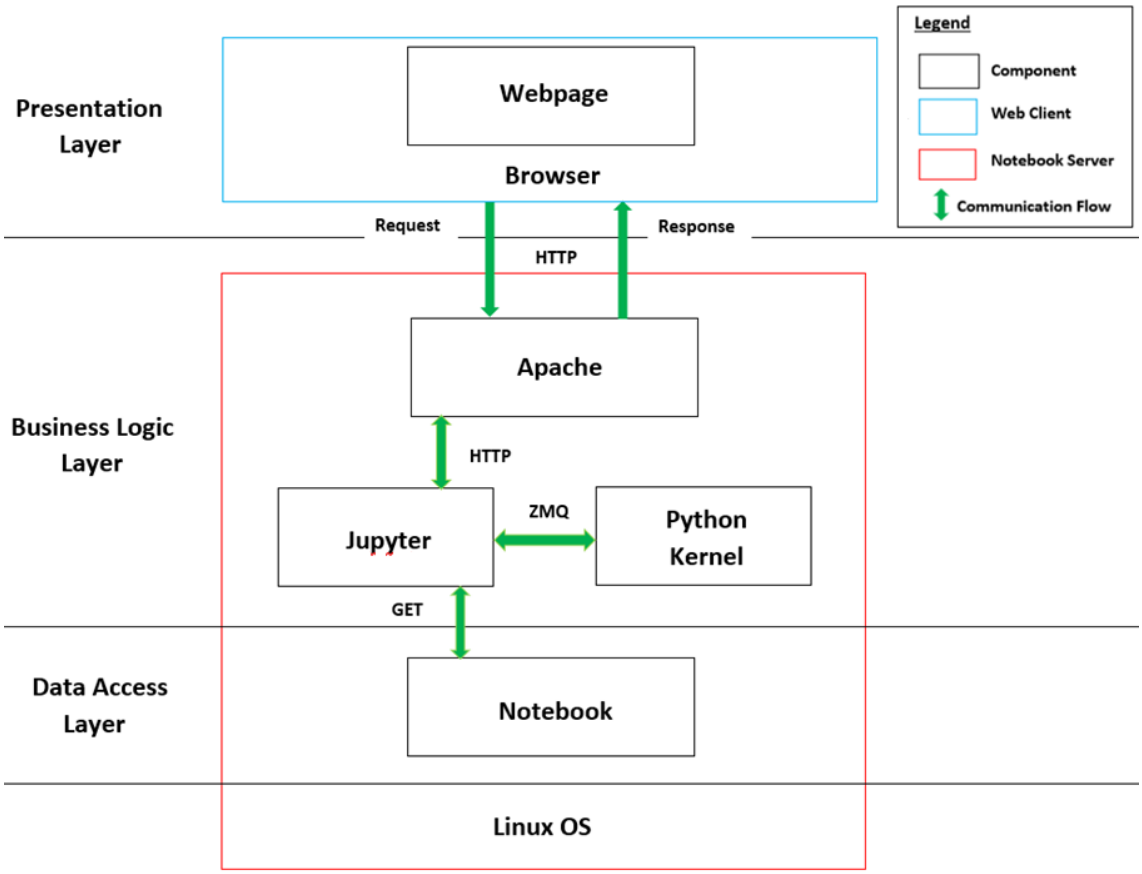

Figure 1.4: JAPyL Distributed Architecture

a standard format that is understood in relational database systems [161].

However, heterogeneous interoperability creates several challenges in multi-tier software stacks including the following as discussed in Section 2.3.2. (1) Interoperating components in multiple languages and paradigms increases developer cognitive load since they must simultaneously reason in multiple languages and paradigms. (2) There must be correct interoperation of components, e.g. adherence to the API or communication protocols between components. (3) Interoperation between different components can lead to diverse modes of failure as each component can fail in unique ways. Many of these challenges are the result of contributing factors like multi-language integration or tight coupling:

(1) Multi-Language Integration — referred to as polyglot programming, some multi-tier software stacks consist of interoperating components in multiple languages and paradigms. A recent survey of 1150 open source projects on GitHub reveals that on average, it is quite common for different software systems to be developed using at least five different languages [94]. This can lead to technical challenges like increased developer effort and semantic friction/impedance mismatch [87].

(2) Tight Coupling — multi-tier software stacks usually require high levels of communication and data marshalling among the various tiers [4]. Components are often highly interde-

pendent on one another. For example, in JAPyL, the Jupyter component is dependent on the Apache component for HTTP requests to be received as shown in Figure 1.4. If one component fails, the entire stack ceases to function as a business unit [158]. Each component is a potential single point of failure (SPOF).

1.3 Techniques for Improving Interoperation

Techniques for improving interoperation in multi-tier software stacks include techniques like microservices and tierless languages. Based on a survey of over 1000 developers from over 45 technology companies, an IBM 2021 report states that 37% of developers are using microservices alongside components within their multi-tier systems. Another 17% have replaced components fully with microservices [66].

These alternatives attempt to solve interoperability challenges through techniques like providing loose coupling of services or eliminating the boundary between the client and server tiers discussed in Section 2.4. However, the use of components in multi-tier software stacks remains common. In 2021, the Apache web server is still utilised in 60% of all WordPress applications hosted online [104].

1.4 Research Questions & Contributions

This thesis investigates techniques to improve heterogeneous interoperability in multi-tier software stacks. The aim is to examine and explore the following research questions:

(1) **To what extent do microservices implementations share similar heterogeneous interoperability challenges as components in multi-tier software stacks?** Some microservices implementations, like chained services, have similar heterogeneous interoperability characteristics to component-based multi-tier software stacks. To what extent do microservice based stacks face interoperability challenges like catastrophic failure?

(2) **How effectively do tierless languages address the challenges of heterogeneous interoperability in multi-tier software stacks?** Tierless languages have proven effective in reducing the number of different interoperating languages and paradigms in web stacks. To what extent can tierless languages reduce the interoperability challenges in IoT stacks?

(3) **To what extent can heterogeneous interoperability challenges be reduced in multi-tier software stacks through tier elimination?** Microservices and tierless languages improve interoperation in multi-tier software stacks to some degree. However, they do not entirely eliminate all factors responsible for heterogeneous interoperability. To what extent can reducing the number of tiers in a component based multi-tier software stack improve interoperation?

The following are the key contributions of this research.

(1) **An investigation into the reliability properties of microservices in multi-tier software stacks.** In the academic and grey literature, microservices are perceived by software architects to offer better reliability than components in multi-tier software stacks because of the loose coupling of services. Chapter 3 investigates the reality of this perceived reliability improvements in two ways. (1) It compares the reliability of independent, standalone microservices in a tiered architecture against a traditional web stack component-based architecture. (2) It combines the established microservice classifications of dependence (individual/chained) and state (stateful/stateless) with reliability (minor/critical): if a minor microservice fails then the application often continues to operate; if a critical microservice fails, the entire application fails. This defines a new three dimensional space: the Microservices Dependency State Reliability (MDSR) classification. Using three web application case studies (Hipster-Shop, Jupyter and WordPress) microservice instances that exemplify the 6 points in MDSR are identified.

A prototype static analyser that can statically identify all six classes in Flask web applications is presented and applied to seven open source applications. It assists in demonstrating that the case study examples in each MDSR class exhibit either a known reliability pattern or a bad smell. The prototype static analyser can statically identify three of six patterns/bad smells in Flask web applications. Hence MDSR provides a structured classification of microservice software with the potential to improve reliability.

The reliability implications of the different MDSR classes are evaluated by running the case study applications against a fault injector to show the following. (1) All applications fail catastrophically if a critical microservice fails. (2) Applications survive the failure of individual minor microservice(s). (3) The failure of any chain of microservices in JPyL & Hipster is catastrophic. (4) Individual microservices do not necessarily have minor reliability implications [123].

(2) **Comparing interoperation in tiered and tierless IoT stacks through the systematic evaluation of four implementations of the UoG smart campus IoT systems.** Internet of Things (IoT) software is notoriously complex, conventionally comprising multiple tiers. The developer must use multiple programming languages and ensure that the components interoperate correctly. A novel alternative utilised in web stacks is to use a single *tierless* language with a compiler that generates the code for each component that ensures their correct interoperation. Tierless languages are used in a variety of software stacks, most commonly for web stacks, however their use for IoT stacks is less common.

Chapter 4 presents the first ever systematic comparison of interoperation in tiered and tierless IoT stacks. This is achieved through the systematic comparison of the imple-

mentations of the prototype UoG smart campus IoT system: two tierless (Clean with iTask/mTask) and two Python-based tiered (Python & MicroPython).

The key results are as follows. (1) Tierless languages have the potential to significantly reduce the development effort for IoT systems, requiring 70% less code than the tiered implementations. Careful analysis attributes this code reduction to reduced interoperation (e.g. two embedded domain-specific languages (DSLs) and one paradigm versus seven languages and two paradigms), automatically generated distributed communication, and powerful IoT programming abstractions. (2) Tierless languages have the potential to significantly improve the reliability of IoT systems, describing how Clean iTask/mTask maintains type safety, provides higher order failure management. (3) The first comparison of a tierless codebase for resource-rich sensor nodes and one for resource-constrained sensor nodes. The comparison shows that they have similar code size (within 7%), and functional structure [87, 88].

(3) **An evaluation of the effectiveness of tier elimination for improving heterogeneous interoperability.** Web applications are structured as multi-tier stacks of components. Each component may be written in a different language and interoperate using a variety of protocols. Such interoperation can lead to a variety of challenges like increased developer effort or catastrophic failure. Alternative approaches do improve interoperability to some degree. However, they do not entirely eliminate all the factors responsible for these interoperability challenges.

Chapter 5 explores a pragmatic approach to reducing web stack interoperation, namely eliminating a tier/component through the integration of the Business Logic Layer and Presentation Layers. That is, the implications of eliminating the Apache web server in a JAPyL 4-tier web stack: Jupyter Notebook, Apache, Python, Linux, and replacing it with PHP libraries in the frontend webpage is explored.

This reveals the following. (1) The JPL 3-tier web stack requires that the developer uses fewer programming languages than JAPyL, i.e two compared with four languages. (2) JPL requires 42% less code than JAPyL. (3) Some functionalities can be automated at the upper levels of the statck due to decreased abstraction. (4) However, the latency in JPL is two to three times greater than that of JAPyL. This shows that the benefits of the tier elimination technique include less developer effort and reduced semantic friction. However, its tradeoffs are high performance overhead & resource consumption and increasing complexity [124].

1.5 Publications & Authorship

Some of the dissertation results have already been published or are awaiting publication in peer-reviewed venues. This dissertation unifies and expands work based on the following publications:

1. Lubbers, M., Koopman, P., Ramsingh, A., Singer, J., AND Trinder, P. Tiered versus Tierless IoT stacks: Comparing Smart Campus Software Architectures. In IoT '20: Proceedings of the 10th International Conference on the Internet of Things (2020), pp. 1-9.

2. Ramsingh, A., Singer, J., AND Trinder, P. Classifying the Reliability of Microservice Architectures. In Proceedings of the 18th International Conference on Web Information Systems and Technologies (2022), pp 21-32.

3. Ramsingh, A., Singer, J., AND Trinder, P. Eliminating Web Stack Components to Improve Interoperability. 2022. arXiv:2207.08019 [cs.PL]

4. Lubbers, M., Koopman, P., Ramsingh, A., Singer, J., AND Trinder, P. Could Tierless Languages Reduce IoT Development Grief? In ACM Transactions on Internet of Things (2022).

The work reported in this dissertation is primarily my own with the exception of Chapter 4 that is collaborative work [87] & [88]. The original University of Glasgow smart campus implementation (PRS) was implemented by Jeremy Singer, Dejice Jacob and Kristian Hentschel using Python on the RaspberryPi [59]. The tierless IoT stacks of CRS and CWS were implemented by Mart Lubbers and Pieter Koopman from the University of Nijmegen.

Chapter 2

Background

This chapter introduces the related literature covering interoperation in distributed multi-tier software stacks. Specifically, it outlines the concept of heterogeneous interoperability in distributed multi-tier software stacks, its challenges and techniques for improving interoperation. Section 2.1 describes conventional software stacks and introduces examples of these stacks. Section 2.2 discusses the similarities between web & IoT stacks. Section 2.3 examines different forms of interoperation in multi-tier software stacks and what factors affect heterogeneous interoperability. Section 2.4 examines alternative techniques for improving interoperation. Finally, Section 2.5 summarises the entire chapter.

2.1 Conventional Web & IoT Stacks

Distributed multi-tier software stacks organise and deploy software components as a hierarchy of interacting tiers. The components are typically heterogeneous, i.e. each component may be written in a different language. The required functionality of the stack is achieved by *interoperating* the components using a variety of protocols.

The following subsections especially Section 2.1.3 and Section 2.1.5 describe distributed multi-tier software stacks that will be used as running examples in this chapter and throughout the thesis.

2.1.1 Conventional Web Tiers

Many web stacks are distributed systems, with client browsers interacting with a remote webserver and data store [92]. For example, in LAMP, the web client consists of the webpage component that interoperates with Apache, PHP and MySQL that are located server side shown in Figure 1.2.

This is a result of the n-tier architecture, shown in Figure 2.1 where the functionalities of a stack are separated into different interacting layers or tiers as follows [93]:

(1) Presentation Layer — the user interface or viewing layer where clients view data or request services via a webpage component.

(2) Business Logic Layer — handles all request processing and marshalling of data.

(3) Data Access Layer — the backend that handles the retrieval and storage of data.

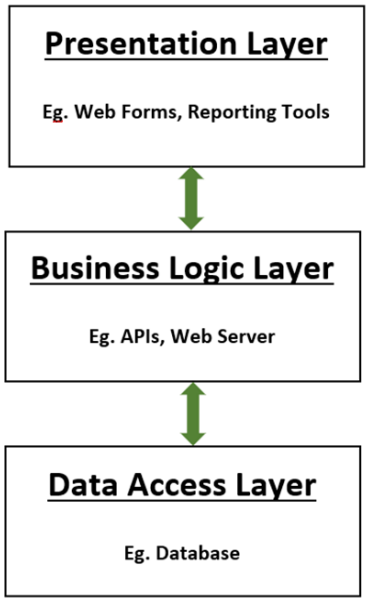

Figure 2.1: N-Tier Architecture

Each tier can interoperate on different nodes or clusters with shared communication. Another option is to have all the different tiers interoperate on a single node [106].

2.1.2 Notebook Programming

Programming environments for conventional web stacks have evolved over time. Classical compilers and editors are still present but are further supported by Integrated Development Environments (IDE) [25]. The next stage of programming evolution is the computation notebook. This is a programming environment that combines text, code and visualisation [132]. It usually consists of the following. (1) A word processor that formats the text. (2) A shell or kernel that executes statements in a programming language. (3) A rendering engine that renders HTML to a user readable format [70]. One example is Jupyter Notebooks.

Jupyter is a distributed system where the client notebook is rendered in a web browser while the server acts as a web host [141]. The application provides support for over one hundred programming languages through the use of a kernel framework shown in Figure 2.2.

Multi-language integration is supported through the use of subkernels. This is a specially created kernel that acts as a proxy or bridge communicator for separate language kernels shown in Figure 2.3. Each cell in the Jupyter Notebook supports its own language.

Figure 2.2: Jupyter Kernel Component

A cell is a multi-line input field for source code execution [116]. Once the code is executed, the subkernel allows the server to route each language to the appropriate kernel and perform the necessary functionality [141].

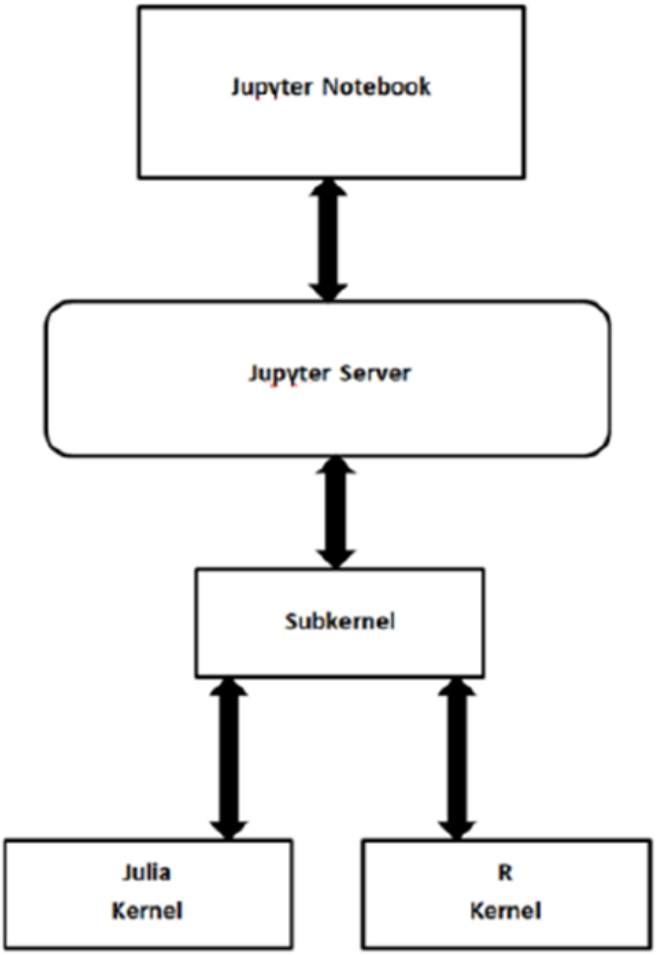

Figure 2.3: Jupyter Subkernel Integration

In contrast to software stacks like LAMP, components in the Jupyter application are loosely coupled to some degree. If programming code is present in the cells of the Jupyter Notebook, successful code execution can still occur even if the Jupyter server is unavailable. However, execution performance of entire code blocks can be slower. The Jupyter application is single threaded and only one cell at a time can be executed in a single notebook [141].

2.1.3 JAPyL

This is a conventional 4-tier architecture composing Jupyter, Apache, Python and Linux (Figure 3.2). In this web stack, Apache is usually interposed between the client and the Jupyter Server as a reverse proxy. It takes requests from clients and forwards them to Jupyter. This is done in order to create an extra layer of security to protect the Jupyter Server [99]. Appendix A.1 provides an example of the configurations a developer has to implement in both Apache and the Jupyter Server to allow for communication between the two components.

The stack features a Defense-in-Depth multi-layered approach to security (Figure 2.4). That is, various security mechanisms are deployed throughout the stack. The intention is that if an attacker targets the Jupyter Notebook online and is able to penetrate one layer, another layer may thwart the attack [99].

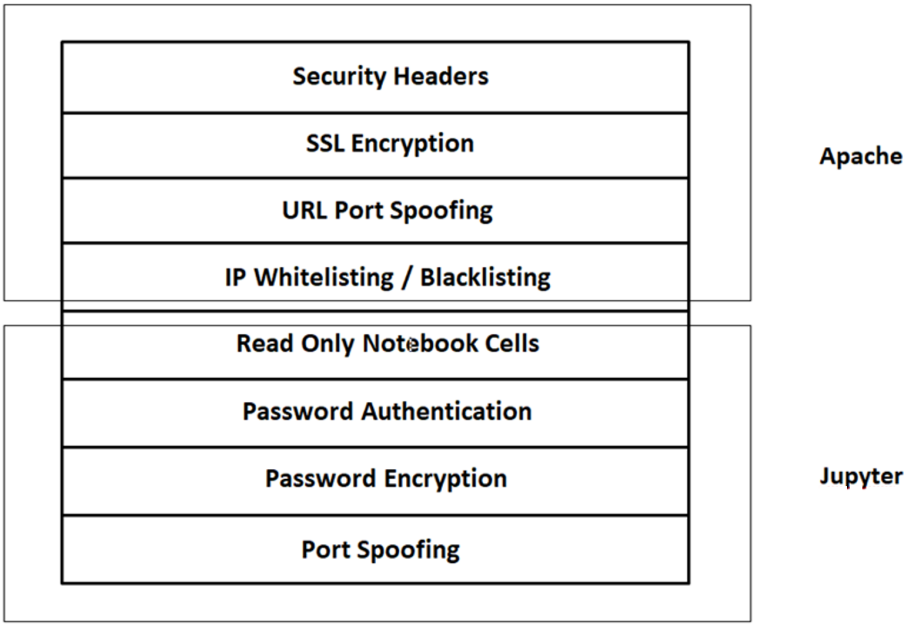

Figure 2.4: JAPyL Defense-in-Depth Security Model

The Apache web server can be hardened to provide several security mechanisms like security headers, SSL encryption, URL port spoofing and IP white/black listing. This secures the web server layer of the JAPyL software stack. The Jupyter Notebook component is secured by the Jupyter Server. Notebook cells can be made read only while access to the data on the notebook itself can be limited via password authentication and encryption as well as port spoofing.

Port spoofing is a technique used by threat actors to bypass network perimeter defenses by sending malicious network traffic through using non standard ports. This attack could go undetected because packets may be going through a port that is not being monitored [58]. However, this technique can also be used as a defense technique. A systems administrator may change standard ports used by an application to send and receive packets through a non standard port [37]. For example, it is common for the Jupyter Server to send and received packets

through port 8888 [99]. However, this port can be changed to any nonstandard port like 10237. A threat actor will not be able to detect or intercept packets from the Jupyter Server if they are monitoring port 8888.

These security mechanisms are typically *declaratively specified* in either the Jupyter Server or Apache [85]. Often security features are implemented as runtime parameters. Appendix B.1 provides examples of some of the declarative security mechanisms used in JAPyL.

Declarative specifications in some domain-specific languages like HTML, XML, or Apache Configs make things easier for the developer. It raises the level of abstraction [115]. Nonetheless, the developer must be able to correctly interoperate several languages, components and APIs. These include (1) Jupyter Configurations (2) Jupyter Components i.e. Kernel, Subkernel & Magic Implementations (3) Different languages like JSON, Apache Configs and Python [99].

2.1.4 Conventional IoT Tiers

Tiered IoT stacks are even more complex than web stacks. They combine a web application that interoperates with a second distributed system of sensor and actuator nodes [87]. This results in a typical IoT stack consisting of at least four interoperating layers shown in Figure 2.5. These are as follows [87]:

(1) Presentation Layer — utilises web components as the interface between the human and devices where application services are provided.

(2) Application Layer — acts as the interface between the Presentation Layer and the Perception Layer, storing and processing the data.

(3) Network Layer — responsible for the communication between the sensor nodes and the server through protocols like MQTT [1].

(4) Perception Layer — collects the data, interacts with the environment, and consists of devices using light, sound, motion, air quality and temperature sensors.

2.1.5 Python Wemos Super Sensor (PWS)

(Micro)Python, Wemos, Super Sensor (PWS) IoT stack is one example of a conventional IoT Stack. The web application components like the webpage or Apache Server are found in the Presentation & Application Layers. Components for the sensor nodes and actuators or the Measurements Collector are found in the Perception & Network Layers shown in Figure 2.6. Each layer can interoperate in its own node.

[1] Source: `http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/os/mqtt-v3.1.1-os.html`

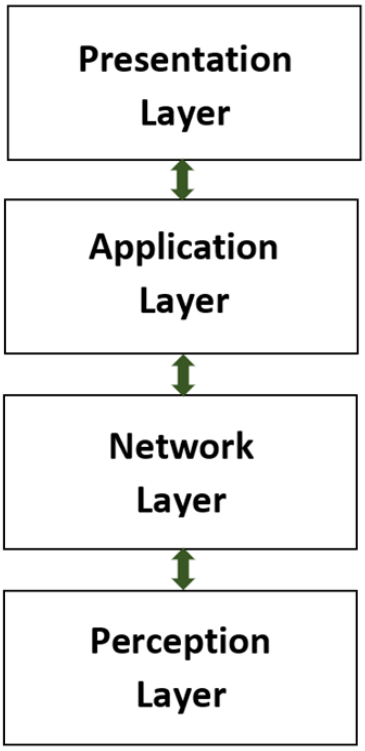

Figure 2.5: A Generic 4-Tier IoT Architecture

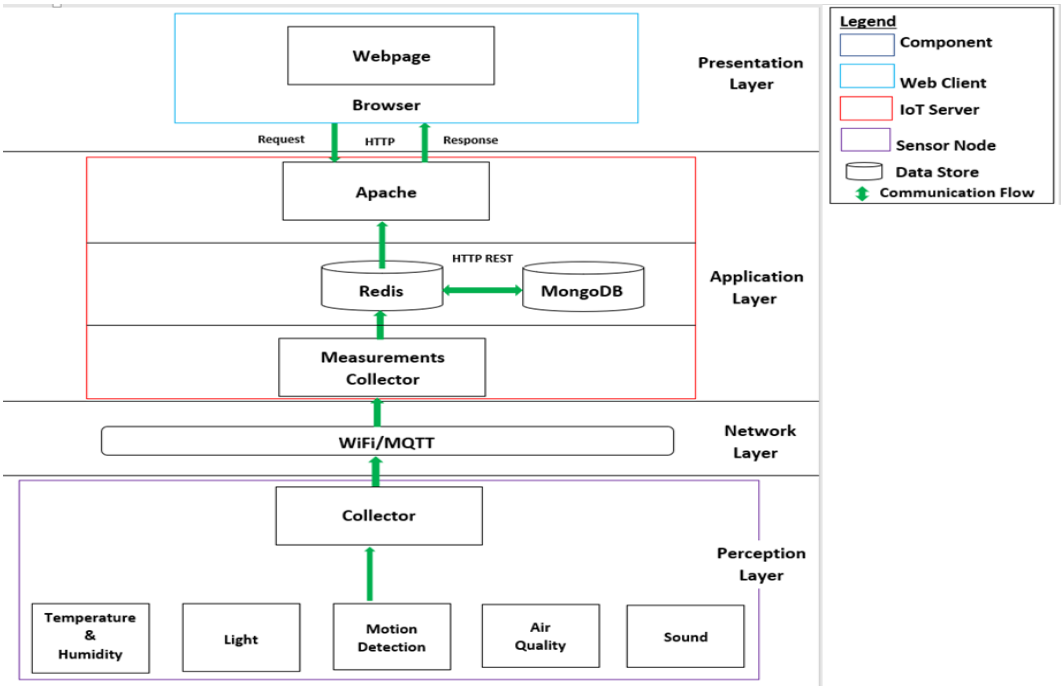

Figure 2.6: PWS Distributed IoT Architecture

2.2 Comparing Web & IoT Stacks

2.2.1 Differences between Web & IoT Stacks

Architecture While both web and IoT stacks are tiered, IoT stacks account for a second level of distribution comprising sensor nodes and actuators that allows the application to collect data from its physical environment. Hence the need for Perception & Network Layers in the IoT 4-tier architecture shown in Figure 2.5

Data Generation In web applications, the content is usually generated by human interaction. One has to input data, ask a query, issue a search or send request for a web service. This form of communication is referred to as human-to-machine communication (h2m) [54].

In contrast, for IoT applications, data is usually generated or pushed through the use of devices like embedded sensors, actuators, processors, transceivers, etc. This form of communication is referred to as machine-to-machine (m2m). There is direct communication between devices using any communication channel that is available especially through wireless means without human involvement [159].

Protocols Generally, some protocols used in web stacks like HTTP are also found in IoT stacks especially in the Perception and Application Layers. However, there are some protocols implemented in IoT stacks that are not usually used in standalone web stacks like MQTT [167].

Figure 2.7 compares the different protocol examples that are utilised in web and IoT stacks when deployed according to the Open Systems Interconnections (OSI) Model [51]. This is a logical and conceptual model that defines network communication as a layered server architecture system. Each layer is defined according to its specific function [83].

2.2.2 Similarities between Web & IoT Stacks

Batory and Malley believe that the building block of any hierarchical platform in any domain is the software component [12]. Many tiered systems are built using similar components no matter the domain [12]. Web and IoT software stacks are an instance of this. For example, tiered web stacks like LAMP consist of data storage, web server and front end components. These are also found in IoT stacks like PWS in both the Perception & Application Layers. There is a backend storage like MongoDB, a front end web page and a web server like Apache shown in Figure 1.3.

The organisation of these components and how they function together are standardised [12]. For example, users interact with JAPyL and PWS via the webpage component. Similarly, a web server like Apache is required to host the webpage component as well as to process HTTP requests shown in Figures 1.3 & 1.4.

[2]Source: `https://www.researchgate.net/figure/`

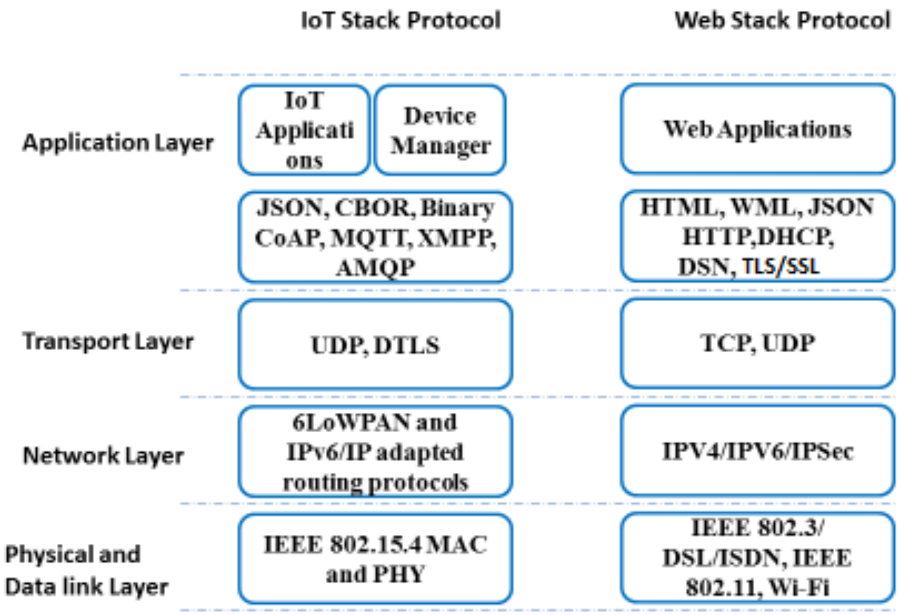

Figure 2.7: Example Comparison of Protocols in Web vs IoT Stacks [2]

As many tiered IoT stacks consist of web components, the n-tier architecture is still present. For example, in PWS, the Business Logic & Data Access Layers are found in the Application Layer while the Presentation Layer acts as the frontend for the entire application [87]. Moreover, the components in both stacks are also usually organised in a client-server architectural style shown in Figure 1.1. The client provides an interface to allow users to either request services or display data. The server handles all the workload. Communication between the client and server usually occurs over a network using standard protocols [86].

Similar components and architectural styles mean that both Web and IoT software stacks share similar forms of interoperation. This will result in similar interoperability challenges for both discussed in Section 2.3.2.

2.3 Heterogeneous Interoperability in Distributed Multi-Tier Software Stacks

2.3.1 Heterogeneous Interoperability

Wegner defines *interoperability* as the "ability of two or more software components to cooperate despite differences in language, interface, and execution platform" [163]. Components interoperate when they are able to communicate, exchange information or share data in a unified format. The meaning of the data being exchanged is understood similarly by all components [65]. Software components have to be able to interoperate both *syntactically* and *semantically*.

Syntactic interoperability involves components being able to correctly read and interpret the

message structure of marshalled data. This requires the use of standard data exchange formats that can be interpreted by all components. Examples include JSON, XML or SQL [6].

Semantic interoperability allows components to understand the exchanged data in similar way without ambiguity or confusion [6]. For example, if the data being marshalled is an integer, all components must be able to understand this data type as an integer. The encoded value should not be interpreted as any other type like string or character [50]. This is a challenge for some IoT applications as some different components may represent their data in different formats that prevent other components from being able to process the data [91].

Syntactic and semantic interoperability are subcategories of heterogeneous interoperability. This is a general term used to refer to all forms of interoperation that occur within a distributed multi-tier software stack [5]. For example, another form of interoperation is *technical interoperability*. This refers to the ability of different components to communicate and exchange data despite utilising different communication protocols [122].

Heterogeneous interoperability exists because many distributed multi-tier software stacks require a variety of networking solutions, protocols, middleware and APIs [15]. For example, in the PWS IoT stack, different components include Python/MicroPython Collectors, MongoDB and Redis. Interoperation is facilitated through a combination of MQTT, HTTP REST & JSON shown in Figures 1.3 & 2.12.

2.3.2 Factors Affecting Heterogeneous Interoperability

Distributed software stacks are no longer hosted on just two or three nodes in a single room. They have grown immensely over time. These stacks can now be hosted on hundreds of nodes across various data centres in several different countries [25].

Successful interoperation of components in these distributed multi-tier software stacks is dependent on how well they are able to communicate and coordinate with one another across the different distributed nodes. *Communication* refers to software components in different nodes being able to exchange messages or marshal data with each other over a network. *Coordination* refers to software components being aware of their specific functionalities and working in sync with other components [139].

Many factors can affect the interoperation of components in distributed multi-tier software stacks. Two of these include coupling and multi-language integration.

Coupling This refers to the degree that software components are interdependent on one another. Different types of coupling exist. Some examples include content coupling, data coupling and control coupling [107].

The type of coupling between components is determined by the number of different modules or API being invoked or shared by different components to marshal data. Basically, the

more modules that have to be shared, the higher the degree of interdependency among components [148].

Good architecture design requires that components have loose coupling. Here, the components should have low levels of dependency on one another or not have high volumes of data marshalling. This is ideally achieved when components are either data or stamp coupled. They only access data elements from other components that are required and the levels of data marshalling is quite low [36]. The main difference between data and stamp coupling is that *data coupling* refers to data that is marshalled between components through a parameter value while *stamp coupling* involves data being marshalled through a record parameter [107].

However, some component-based software stacks like LAMP, JAPyL or PWS are usually highly interdependent or tightly coupled. This is due to *control coupling* as components have to request and marshal data between each other. Each component requires this data in order to execute its required functionality and for the entire stack to achieve its overall business function. The levels of communication, coordination and data marshalling are very high [79].

Furthermore, each component in the stack has to share similar libraries, code bases or even processes to ensure that the meaning of data being passed is preserved [4]. For example, in PWS, Redis & MongoDB components share data by accessing the same JSON library in the Python cache module shown in Listing 2.1.

Listing 2.1: Redis & MongoDB Accessing Similar Python Module

```
1
2  import redis
3  import json
4  from bson.json_util import dumps
5  from pymongo import MongoClient
6
7  client = MongoClient('')
8  redisClient = redis.StrictRedis(host='', port=6379, db=0)
9
10 database = client['supersensorvalues'] #database name in mongodb
11
12 sensordataList = database['sensor_data'].find() #collection name in database
13
14 serializedObj = dumps(sensordataList) #serialize object for the set redis.
15 result = redisClient.set('sensordata', serializedObj) #set serialized object
       to redis server.
16
17 parsedsensordataList = json.loads(redisClient.get('sensordata'))
18
19 for sensordata in parsedUserList:
20  print(user["sensor_type"])
```

Interoperating components communicate synchronously in many distributed multi-tier soft-

ware stacks. They need to know the state of each other in real time before coordination and sharing of messages occur. Components tend to remain idle until they receive a request or message from other components [134]. If a component fails, others will continue to remain idle as they are not receiving any requests or status updates from the failed component.

Tight coupling can lead to *catastrophic failure*. This is the complete, sudden and often unexpected breakdown in the behaviour of a system. From a systems perspective, catastrophic failure occurs when there is a critical interruption that causes a component or application to fail [105]. However, if the user cannot access the application frontend due to challenges like the application being unable to accept HTTPS requests, then the system has also failed catastrophically.

Multi-Language Integration Programming languages are usually categorised into *paradigms*. This is a programming language style used to develop an application. Some examples are as follows. (1) The imperative paradigm is where the programming code describes the steps necessary to perform a computational task. (2) The object oriented paradigm is where computations are performed through the passing of messages to objects. (3) The functional paradigm involves computations being performed through functions. (4) The declarative paradigm involves a computational task defining what needs to be done without describing the control flow [25]. Table 2.1 provides examples of some programming languages and their paradigm classification.

Many programming languages are *multi-paradigm*. They can be used to develop an application using a combination of these different programming styles [168]. One example is the use of F# and Scala first class functional concepts that have been added to mainstream object-oriented languages like LINQ extensions to C# [25].

Polyglot programming is part of the evolution of programming languages and software development. It refers to the development of software or applications using multiple programming languages and paradigms. It facilitates language interoperation and allows developers to add functionalities that may not be readily available in a single language [140]. For example, in JAPyL, the programmer utilises four languages in the implementation of the stack discussed in Section 5.4.2. Later, in Section 3.5.1, it will be shown that Hipster-Shop utilises ten programming languages.

Table 2.1: Language Paradigm Classification

Language	Paradigm
Python	Imperative, Object Oriented
PHP	Imperative, Object Oriented
C++	Imperative, Object Oriented
Java	Object Oriented, Event Driven
JavaScript	Imperative, Event Driven
Go	Imperative, Functional

In tiered software stacks, an inverse pyramid classification can be used to categorise the mul-

tiple languages implemented at three different layers. The Domain Layer classifies languages that are Domain Specific Languages (DSLs) like HTML or CSS, the Dynamic Layer includes all dynamically typed languages like Python, Scala or JavaScript and the Stable Layer classifies languages that are statically typed like Java or C++ [73]. Figure 2.8 provides an example of the inverse pyramid classification for languages utilised at the different layers in a typical tiered web application. The pyramid itself represents the amount of time the original code base could possibly remain unchanged in an application. The longer the code is expected to remain unchanged, the closer it will be to the bottom of the pyramid [9].

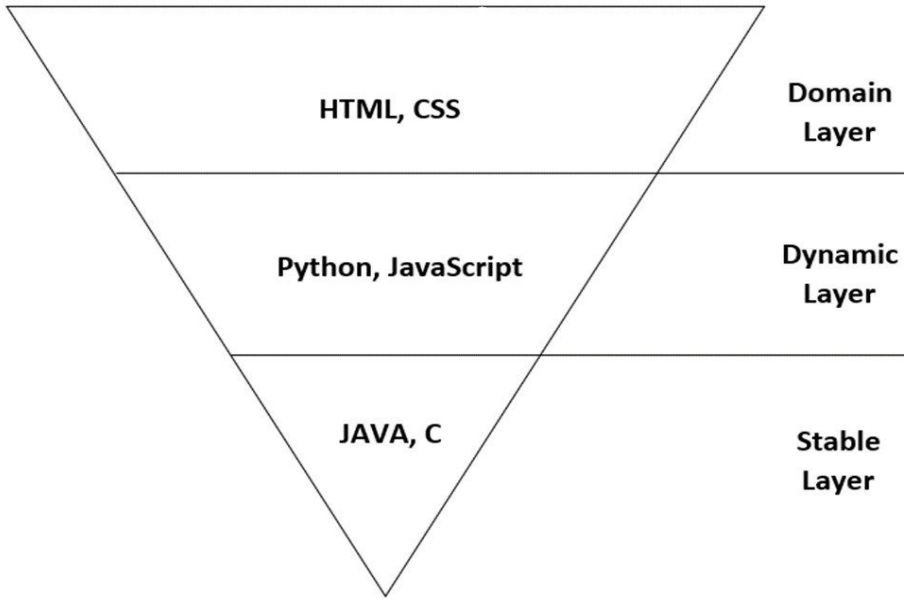

Figure 2.8: Inverse Pyramid Classification[3]

The integration of different languages especially in distributed multi-tier software stacks can lead to several interoperation challenges like:

(1) Polyglot Development — the developer must be fluent in all the different languages in the stack and be knowledgeable in how to make them interoperate together. This person is referred to as a full stack developer [96].

(2) Correct Interoperation — the developer must be able to correctly use multiple languages and ensure they adhere to the API or communication protocols between components [67].

(3) Maintaining Type Safety — a key element of semantic friction. This refers to all the mismatches that can arise from integrating different languages. Ensuring type safety is crucial for correctness but the diversity of type systems in different languages makes this difficult [67].

[3]Source: [73]

(4) Performance/Resource Overhead — inter-process communication is typically slower than a function call as data must be marshalled for communication between languages and components [11]. The reason is that time is either needed for the different languages to communicate with one another or to translate messages from one language to the next [82].

2.4 Techniques for Improving Interoperation

2.4.1 Reducing the Number of Languages

Some techniques attempt to improve interoperation by focusing on reducing the number of programming languages. This can be achieved through stack implementation, creating a runtime, reducing the number of APIs, combining multiple languages into a single programming language and browser compilation.

Single Stack Language Polyglot programming can lead to challenges like increased developer effort or semantic friction in multi-tier software stacks. The use of a single programming paradigm consistently throughout the stack reduces some of these challenges. It can eliminate semantic friction and improve data marshalling [106].

ASP.NET is an example of a web programming language that can be used to develop web applications without the need for implementing client side languages like HTML or CSS. It is a server side object oriented language that can manipulate images, shapes, text boxes and even text as well as perform business logic processes like creating session states [156]. Listing 2.2 provides an example where ASP.NET code is used to define the size, border and color of a table and image object without having to use HTML or CSS.

Listing 2.2: Example of ASP.NET Code

```
1
2 <asp:TableCell BorderStyle="Groove" BorderColor="#0000ff" BorderWidth="1px">
3
4 <asp:HyperLink ID="HyperLink1" runat="server" NavigateUrl="Default.aspx"><
    asp:ImageID="Image1" runat="server" ImageUrl="images/logo.jpg"
    AlternateText="Picture of Vision Logo" Width="387px" Height="110px" />
5
6 </asp:HyperLink>
7 </asp:TableCell>
```

JavaBeans is an example of a stack that utilises a single language in developing web applications. The stack usually consist of Java based components like Enterprise Java Beans (EJB), an EJB container and a Java Application Server [48]. When the client is invoked, the EJB container manages resources for the Java Application Server while the Enterprise Java Beans component handles client requests shown in Figure 2.9.

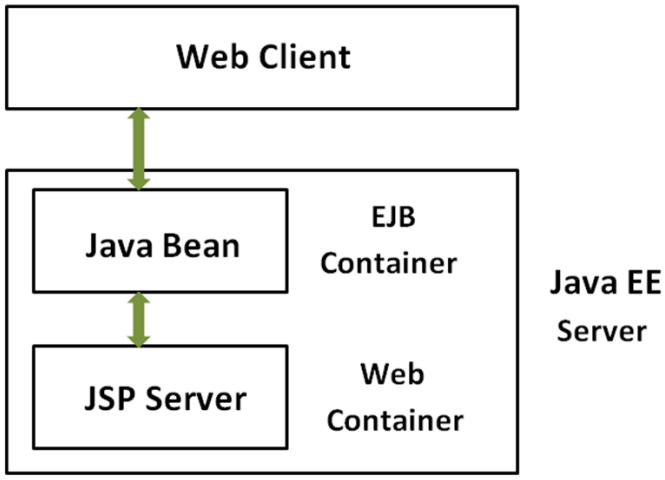

Figure 2.9: Java Beans Architecture

However, a more popular option is the MEAN web stack (Figure 2.10). It uses JavaScript as the primary programming language and composes MongoDB, ExpressJS, AngularJS & NodeJS [31]. This language is now considered one of the most utilised programming languages on GitHub and the most tagged in StackOverflow [25].

MEAN typically outperforms Linux, Apache, MySQL & PHP (LAMP) by two times or more due to the computational inefficiencies of PHP, and the inefficiency of Apache in handling I/O operations [24]. However, the stack is prone to catastrophic failure due to tight coupling. The client side is browser dependent [31]. If the JavaScript module is disabled in the web browser, the client side will not function. The application is rendered unusable to an end user even if the server components continue to function.

Runtime Environment Runtimes or Virtual Machines (VM) minimise interoperation overheads between languages. They are an intermediate concept between a compiler and an interpreter. The idea is to write program code in a high level language that is then translated to an intermediate language referred to as bytecode [49]. This is shown in Figure 2.11.

Java Virtual Machine (JVM) is one example of a runtime. It is available for a variety of platforms and allows for several programming languages like Clojure, Ruby, Scala and Python to be converted to Java bytecode. This is beneficial as it allows developers to interoperate different languages with Java libraries without having to adopt a new language [82].

Another example is the .NET Framework that typically composes WinForms, ADO.NET, Framework Class Library and the Common Language Runtime (CLR) [49]. Having the CLR as a common runtime facilitates the interoperation of a number of Microsoft technologies like ASP.NET, VB.NET, C#, F#, etc. However, .NET is proprietary and only available on the Windows Operating System.

A challenge for runtime environments is performance overhead especially in code execution. For example, a newer runtime is the Truffle framework. It interoperates JavaScript, Ruby,

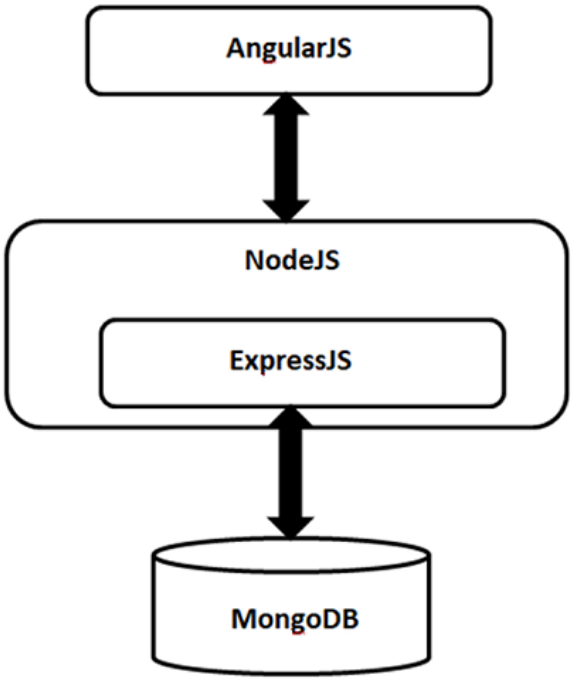

Figure 2.10: 4-Tier Single Language MEAN Architecture

Python, JAVA, Scala, C++, etc. It does so using the GraalVM compiler and an abstract syntax tree interpreter. In order to perform functions like string comparison and pattern matching, the platform has to perform reference equality checking and arity checking that can slow down compilation and incur some overhead [50, 157].

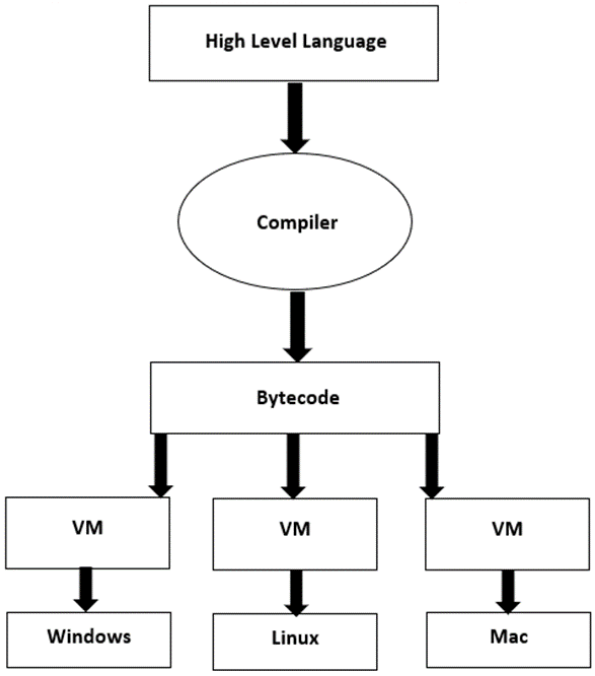

Figure 2.11: Virtual Machine Code Translation Process

Single API The Application Programming Interface (API) is a defined set of rules that allow different components to communicate with one another. It acts as an intermediary layer that facilitates the exchange of data in a unified format that is understood by all components in the application [108].

In some conventional multi-tier software stacks, it not uncommon for several different API protocols to be utilised. These can take several different forms like SOAP, XML, JSON or REST [127]. For example, in the PWS IoT stack shown in Figure 2.12, both JSON & HTTP REST protocols are present in the Application Layer.

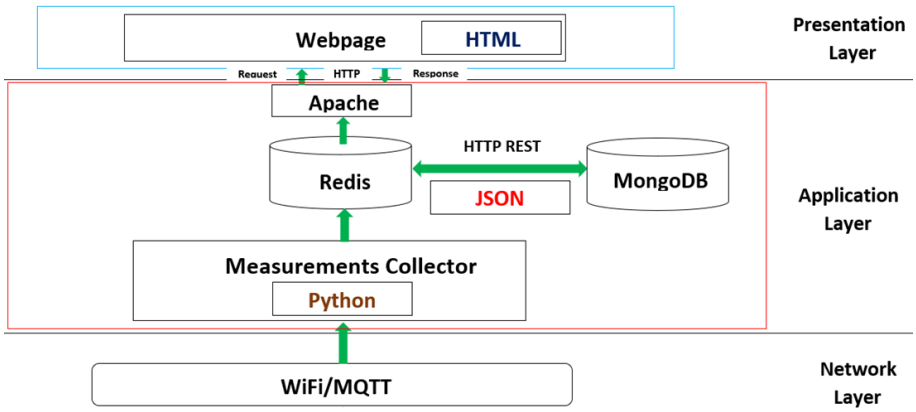

Figure 2.12: API Protocols in PWS Application Layer

The issue with integrating too many different APIs is performance. Generally, APIs are designed to process data quickly. However, if the system does not assign the necessary resources for processing data like memory or bandwidth, the response time will be slower. Too many different forms of API can put a further strain on system resources [127].

One example for reducing the number of APIs in IoT stacks is the use of a single language IoT system based on the RIOT Operating System (OS). Here, both client and server are written in JavaScript [7]. All communication between the physical components like sensors and the software components in the cloud is facilitated by a single JavaScript API.

Combining Languages Some programming languages can be composed together to create a new language. One example is PyHyp where both Python and PHP are fused together. A single module can consist of code from both languages shown in the example in Listing 2.3. Here, the elements in the PHP array are passed to Python as an object. HippyVM translates the high level code to bytecode [11].

Listing 2.3: PyHyp Application Example

```
1
2 import sys
3
4 $sport = array("Football", "Cricket", "Basketball", "Volleyball");
```

```
5 $result = json_encode($sport);
6
7 result = sys.argv[1]
8 print(result)
```

As both languages are composed together into a single language, semantic friction is reduced. However, performance overhead is two times slower compared to its mono-language constituents [11]. Moreover, PyHyp is a relatively immature technology and is not regularly updated.

Browser Compilation Here, components are written in a single programming language and then compiled to another language to be executed in the browser. Two examples include (1) Google Web Toolkit where Java is used for high level language development but is then compiled to JavaScript [34]. (2) Web Assembly where the developer utilises languages like RUST that is compiled to bytecode and then executed in the browser automatically [55]. Given that the languages are compiled, execution speeds and interoperation performance will be faster compared to interpreted bytecode.

One of the main issues with this approach is that some functionalities or services are browser dependent (tight coupling). For example, while web assembly can be a compiler target for languages like C or C++, it can only interact with its internal environment. External functions like printing text requires that web assembly call a function that the browser provides [55]. If that function is disabled in the browser settings or the browser component itself is non functional, web assembly cannot perform the requested service.

2.4.2 Microservices

Microservices are fine grained code services that are intended to be loosely coupled and independently deployable. The architecture involves developing an application as a suite of small services. Each one runs independently in its own process and communicates with lightweight mechanisms, usually via RESTful web services or messaging. The idea is to replace components in a stack with lightweight code services [133].

Loose coupling is usually achieved in microservices through data coupling. The rate of communication or data marshalling with other services is very low [133]. Moreover, microservices usually express computations at a high level of abstraction because they have their own in-built runtimes, functionalities and data stores. Functionality can be implemented with just a few lines of code. There is decreased reliance on third party libraries that can span numerous modules [57].

There are two options for migrating to a microservices architecture:

(1) Convert a few of the components to microservices — both components and microservices can interoperate with each other in distributed multi-tier software stacks. For example,

in the JAPyL web stack, ZMQ messages from the Python kernel component are necessary for the Python language to communicate with the notebook being hosted as shown in Figure 1.4. However, the use of the kernel component can be eliminated locally in the stack using a microservices architecture. A Jupyter Kernel Gateway service can be implemented that provides headless access to a Jupyter Python kernel service stored in a remote location. Communication with the remote kernel service relies on REST calls rather than ZMQ messaging [99].

(2) Convert all components to microservices — the distributed multi-tier architecture will consist of tiers of microservices [160]. One example is the Layered Microservices Model (LMM) shown in Figure 2.13 . Tiered microservices are also utilised in some IoT Stacks where the software components are fully replaced with tiers of Device or Tenant Microservices [146].

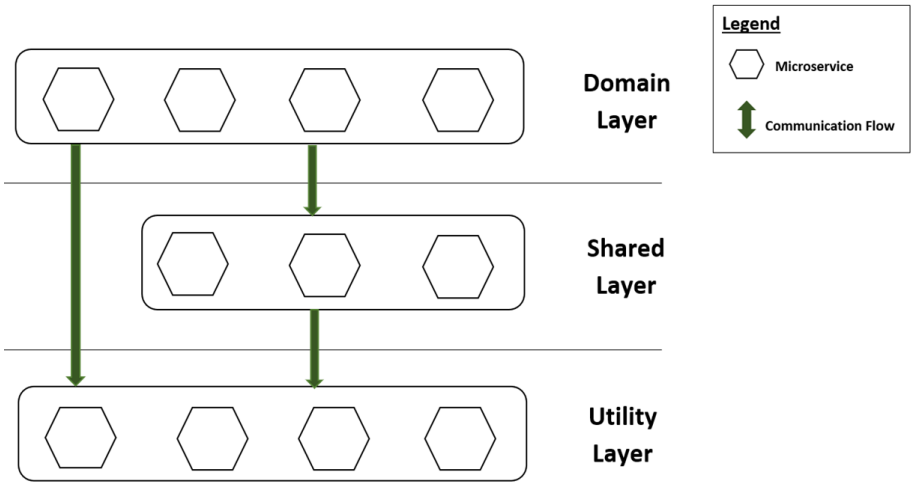

Figure 2.13: Layered Microservices Model

Structuring software using the microservices architecture is a popular trend. Technology companies like Netflix, Amazon & Spotify have adopted it in the implementation of their applications & services [150].

Replacing components with microservices in distributed multi-tier software stacks should improve reliability. As each service is independent or loosely coupled, interoperation challenges like catastrophic failure should be reduced [133]. However, some types of microservices exhibit heterogeneous interoperability characteristics like tight coupling. This makes the architecture prone to catastrophic failure similar to that of components in multi-tier software stacks [123]. This will be discussed in Section 3.9.4.

2.4.3 Tierless Languages

Tierless languages usually incorporate the use of a single language that synthesises all components/tiers in the software stack. The developer usually writes the entire stack as a single application. The overall objective of this paradigm is to eliminate the boundary between the client and server tiers [87].

Interest in developing tierless or multi-tier languages has been growing over time. Many early attempts like PL/I proved fruitless because of the complexity in trying to get a single high level language to compile or translate into different languages with different programming paradigms [25]. However, with the improvement in computing resources and language evolution, researchers have developed a number of tierless languages including Links, Hop and Haskino [27, 33, 135]. These languages are also being incorporated into many industrial technology applications e.g. [8, 14, 144]. They offer significant benefits like less development effort, better maintainability and sound semantics of distributed execution.

Tierless Classification Tierless languages have been developed for a range of distributed paradigms, including web applications, client-server applications, mobile applications, and generic distributed systems. These languages can be classified based on a number of criteria like compilation scheme, distribution architecture support or remote communication techniques.

Tierless languages like Hop.js utilise a uniform compilation scheme. This is where the language is compiled to the same language on both the client and server side. [164]. In contrast, languages like Links compile to different languages on both the client and server side. This is referred to as cross compilation. Table 2.2 provides examples of some tierless languages that support uniform or cross compilation.

Table 2.2: Examples of Tierless Languages based on Compilation Scheme.[4]

Language	Uniform Compilation	Cross Compilation
Fun	x	-
Koka	x	-
Hop	-	x
Links	-	x
Opa	-	x

Many tierless languages like Links or Hop support the client-server architectural style for distribution. However, some of the languages are flexible and provide support for other distribution architectures. For example, AmbientTalk/R can support both the client-server and peer-to-peer distribution architectural styles [164].

Tierless languages utilise techniques similar to mainstream programming languages to facilitate remote communication. These include remote procedures, message passing, publish-

[4]Source: [164]

subscribe, reactive programming and shared state. However, in contrast to mainstream programming, tierless languages attempt to abstract these techniques to make these remote communication implementation easier. Abstraction involves either using syntactic constructs, customised libraries or emulating data structures [164]. Table 2.3 provides examples of some tierless languages and the remote communication techniques they support.

Table 2.3: Examples of Tierless Languages based on Remote Communication Techniques.[5]

Language	Remote Procedures	Message Passing	Publish Subscribe	Reactive Programming	Shared State
Fun	-	-	-	-	x
Koka	-	x	-	-	-
Hop	x	x	x	-	-
Links	x	x	-	-	-
Opa	x	x	-	-	-

Tierless Web Languages The use of tierless languages in web stacks is well established. Languages like Links [27] or Hop [135] are common examples. Here, the code for different tiers is simultaneously checked by the compiler and compiled to the required component languages. For example, Links compiles to HTML & JavaScript for the client side and to SQL on the server side in order to interact with the database system shown in Listing 2.4.

Listing 2.4: Links Compilation to SQL

```
1
2  #Links Code
3
4  fun formatDef(def) client {
5  <span l:onclick="{editDef(def)}">
6  {stringToXml(def.word)}
7  {stringToXml(def.meaning)}
8  </span>
9  }
10
11 take(10, for (var def <-- defsTable)
12 where (def.word    /s.*/) orderby (def.word)
13 [def])
14
15 #Translated to SQL Code
16
17 SELECT def.meaning AS meaning, def.word AS word
18 FROM definitions AS def
19 WHERE def.word LIKE    {s}%
```

[5] Source: [164]

```
20 ORDER BY def.word ASC
21 LIMIT 10 OFFSET 0
```

Another example is Haste. It utilises a tierless web framework and is embedded as a domain specific language (DSL) in Haskell [35]. Haste programs are compiled multiple times. Here, the server code is generated by the standard GHC Haskell compiler; JavaScript for the client is generated by a custom GHC compiler backend. The design leverages Haskell's high-level programming abstractions and strong typing, and benefits from GHC: a mature and sophisticated compiler [56].

Potentially, tierless languages in web applications are beneficial as they improve type safety and enable automatic communication. However, there are some practical challenges. For example, Appendix B.2 shows an instance where Links requires additional developer effort compared to PHP and MySQL.

Tierless IoT Languages The utilisation of tierless languages in the implementation of IoT software stacks is more recent. For example, Erlang or Elixir is now being utilised to engineer IoT systems in production environments. The language is considered tierless because there are sets of Erlang processes distributed throughout the Perception, Network and Application Layers. This language is mostly utilised on resource-rich nodes [88]. However, many of these systems are not referenced in the academic literature with exceptions to [138, 142]. Perhaps, this may be due to the challenges of implementing tierless languages in distributed IoT software stacks [88].

One major challenge is that compilation of the Perception Layer can be slow on resource constrained microcontrollers. Memory is usually limited on these devices. For example, Potato [32] can be implemented as a Functional Reactive Programming (FRP) tierless IoT language on resource-rich nodes [33]. However, it is a memory intensive language and is not suitable for resource constrained microcontrollers.

Program splitting requires that tierless languages automatically determine which parts of the program correspond to the appropriate tier. For example, a tierless web language must be able to identify the relevant code on both client and server tiers and compile both almost simultaneously [88]. This requires the developer to be familiar with many program splitting techniques like syntactic markers or type translation to implement automatic and accurate compilation [27]. Moreover, many sensor nodes on microcontrollers can only be programmed by writing the code to flash memory. The developer has to put in extra effort and ensure there is physical access to the microcontroller to make programming updates [88].

Dynamic provisioning using techniques like over-the-air programming makes maintainability of microcontrollers easier. The technique reduces the number of languages needed for development of the IoT application. For example, JavaScript alone can be used to program both the client and server nodes as in the case of the RIOT IoT System discussed in Section 2.4. However, the RIOT IoT system is not considered tierless as the overall goal of tierless languages is

to integrate both the client and server tiers [164].

Security is a major issue for IoT systems. All layers of the 4-tier software stack are vulnerable to some extent. The Presentation and Application Layers are vulnerable to attacks like SQL Injection. However, these attack vectors can be eliminated using standard web application defense techniques. Defending the Network and Perception Layers is even more challenging. Microcontrollers lack the resources to enable many security mechanisms like encryption. They are also difficult to patch because the program and operating system are stored in flash memory [88].

Tierless languages enhance the security of IoT systems by performing type checking at the time of compilation. This can minimise vulnerabilities like SQL Injection. Some tierless languages like Swift and Jif/split place components to protect the security of data [26, 169]. However, many tierless languages like Erlang lack many important security measures that leave IoT applications vulnerable to threat actors [155].

2.5 Chapter Summary

This chapter outlines the research literature and some technologies related to the work presented in this thesis. It has reviewed the different forms of interoperability and how they relate to heterogeneous interoperability in distributed multi-tier software stacks.

The interoperation challenges are similar for both Web & IoT software stacks. Some of these include catastrophic failure, semantic friction, increased developer effort and type mismatches. Many problems can be attributed to tight coupling and multi-language integration.

Several attempts have been made to improve interoperation in distributed multi-tier software stack. These include the utilisation of techniques like reducing the number of languages implemented in distributed software stacks (Section 2.4.1), introducing notebook programming (Section 2.1.2), microservices implementation (Section 2.4.2), browser compilation (Section 2.4.1), combining programming languages (Section 2.4.1) and using tierless languages (Section 2.4.3).

Each technique improves interoperability to some degree but does not solve all the challenges. Interestingly, some of these techniques like microservices display similar heterogeneous interoperability challenges as components in distributed multi-tier software stacks. Focus on this will be investigated in greater detail in Chapter 3.

Chapter 3

Classifying the Reliability of Microservices

Microservices are popular for web applications as they offer better scalability and reliability than monolithic architectures. Reliability is improved by loose coupling between *individual* microservices. However in production systems some microservices are tightly coupled, or *chained* together.

This chapter classifies the reliability of microservices: if a minor microservice fails then the application continues to operate; if a critical microservice fails, the entire application fails. Combining reliability (minor/critical) with the established classifications of dependence (individual/chained) and state (stateful/stateless) defines a new three dimensional space: the Microservices Dependency State Reliability (MDSR) classification.

Using three web application case studies (Hipster-Shop, Jupyter and WordPress) microservice instances are identified that exemplify the six points in MDSR. A prototype static analyser is presented that can statically identify all six classes in Flask web applications. It is applied to seven applications in this study.

The case study microservices implementations exhibit either a known reliability pattern or a bad smell. The prototype static analyser can identify three of six patterns/bad smells in Flask web applications. Hence MDSR provides a structured classification of microservice software with the potential to improve reliability.

Finally, the reliability implications of the different MDSR classes are examined by running the case study applications against a fault injector. (1) All applications fail catastrophically if a critical microservice fails. (2) Applications survive the failure of individual minor microservice(s). (3) The failure of any chain of microservices in JPyL & Hipster is catastrophic. (4) Individual microservices do not necessarily have minor reliability implications.

This chapter is structured as follows. Section 3.1 describes the motivations for this research. Section 3.2 examines the reasons for failure in component based multi-tier software stacks and why microservices are perceived to improve reliability. Section 3.3 evaluates whether microservices do improve reliability in component based multi-tier software stacks. Section 3.4 describes the contributions of this research in examining the reliability properties of microservices. Sec-

tion 3.5 describes the microservices case study applications that are being evaluated. Section 3.6 describes the different properties of microservices. Section 3.7 describes catastrophic failure in micoservices with different properties. Section 3.8 provides details on the MDSR Classification. Section 3.9 evaluates the findings of MDSR. Section 3.10 summarises these findings. Finally, Section 3.11 summarises the entire chapter.

3.1 Motivation

Microservices are a recent and popular software architecture trend. They are available to replace some or all components in multi-tier software stacks [143]. Some microservices are *stateful*, recording data, e.g. participants in a web chat. Others are *stateless*, i.e. they simply accept requests and purely process them.

Component-based multi-tier software stacks are prone to *catastrophic failure*, where user-visible functionality is suddenly and permanently unavailable as discussed in Section 2.3.2. It is common for the failure of a single component to cause the entire system to fail (a cascade failure).

In contrast, the microservices architecture potentially improves interoperation as it provides *improved reliability* via the loose coupling of services. If one microservice fails, others will remain available [69]. A failure may cause a reduction in throughput but will most likely avoid catastrophic failure. In the worst case scenario, the loose coupling of services enables graceful failure [143].

This is achieved based on the design principle that microservices are implemented as standalone, independent services [68]. However, many large scale web applications include *chains* of microservices where a set of services are closely dependent, e.g. the Netflix Titus Platform [89].

Chained microservices are tightly coupled, e.g. by high-frequency API-based interaction sequences. For example, in Figure 3.1, the Reverse Proxy service is dependent on constant communication with the Port Config service in order to determine what port to use.

Chained microservices make an application far less reliable because if any of the services fail the entire chain fails, and may induce catastrophic failure [60]. For example, in 2014 BBC experienced a critical database overload that caused many of its critical microservices to fail one after another [28]. In 2015, Parse.ly experienced several cascading outages in its analytics data processing backend due to a microservices message bus overload [101].

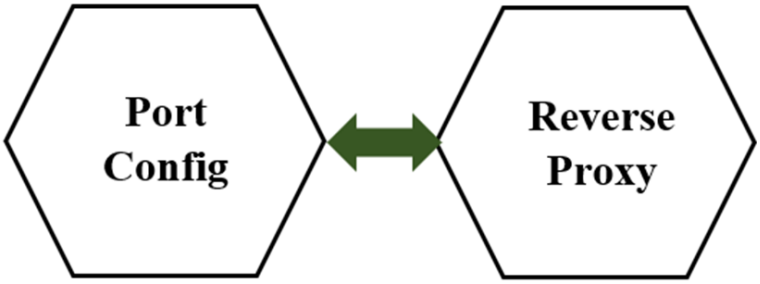

Figure 3.1: A Chained Microservice Example

3.2 Failure in Multi-Tier Software Stacks

3.2.1 Tight Coupling in Component Based Multi-Tier Software Stacks

Conventional multi-tier software stacks like Jupyter, Apache, Python & Linux (JAPyL) comprise multiple tiers of components shown in Figure 1.4. Each tier fulfils a well defined function [158], and the architecture is modular as a tier may be replaced with a component with similar functionality, e.g. there are many suitable relational DBMS or web servers. However, components in multi-tier stacks are tightly coupled as they are highly dependent on one another for the stack to function.

Tight coupling among components in the stack is caused by control coupling discussed in Section 2.3.2. Every component becomes a potential single point of failure (SPOF). If one component fails, the entire business function of the stack is significantly interrupted and results in catastrophic failure [105]. For example, in PWS (Figure 1.3), if either of the collector components fail, the data collected from the sensor nodes in the Perception Layer cannot be stored in either the MongoDB or Redis databases in the Application Layer. The webpage component in the Presentation Layer will not be able to display the data being collected by the sensor nodes. The application is considered to have failed catastrophically because users cannot view the relevant data.

Similarly, in JAPyL, if either the Jupyter or Apache components fail, both will be unable to communicate or marshal data between each other. This will result in catastrophic failure as the entire stack will cease to function as a business unit. The stack will no longer be able accept HTTP requests or users will not be able to view any data from the Notebook via the webpage component.

3.2.2 Microservices Reliability

To mitigate against catastrophic failure, Lewis and Fowler propose the following three design principles for the microservices architecture [81]:

(1) Independent services — each service should run in its own process and be deployed in its

own container like one service per Docker container.

(2) Single functionality — one business function per service. This is referred to as the Single Responsbility Principle (SRP).

(3) Communication — should be via a REST API or message brokers.

Others have added other principles like reliability. This principle states that a microservice should be fault tolerant so that in the case of failure, its impact on other services will be negligible [119]. This can be achieved using design patterns like database-per-service, timeouts, bounded retries, circuit breakers and bulkheads to tolerate failures [60].

There are substantial studies of the reliability of microservice software in both the academic [60, 153, 170] and grey literature [52, 165]. These reveal that reliability in the microservices architecture is not always attained because the reliability design principle is not always followed.

Developers do not always implement the necessary design patterns to prevent microservices failure. Even if they do, they often remain unaware whether their microservice can actually tolerate failures until it actually occurs [60]. Thus, the impact of a microservices failure on an application is not always readily known beforehand.

3.2.3 Partial Failures in Microservices

Even if the microservices design principles are followed, failure is to be expected for two major reasons:

(1) Functional Failures — occur due to poor implementation e.g. "A SQL column is missing" error is returned upon some data request.

(2) Environmental Failures — due to misconfiguration of the infrastructure necessary to run the microservices effciently e.g. microservices processing of requests is slow due to insufficient memory being made available in the Docker environment.

The literature distinguishes partial and catastrophic failures in microservices. Some partial failures may be temporary and recovery is usually automatic, e.g. a microservice without a load balancer may be briefly overloaded [170]. Downtime can often be minimised if replacement microservice instance(s) are activated automatically [153]. Partial failure is considered acceptable in the design of microservices applications. It will be shown in Section 3.6.3 that partial failures should be distinguished from minor failures.

3.2.4 Identifying Causes & Detecting Failures in Microservices

Failures in microservice-based applications may arise from the microservices, or from the infrastructure services like libraries, containers like Docker, development frameworks like Flask, etc. This study focuses on failures in the microservices, and detecting these failures is often challenging.

A common approach is to configure a collection of microservices indicators (SLOs) to continuously monitor for the causes of failure. The SLOs are typically time series, e.g. the response time of a microservice to requests from other services. The microservice is identified as failing if it fails to meet the expected SLO [97].

Service Dependency Graphs (SDGs) can be used to dynamically detect microservices bad smells by mapping their node relationships [89]. However, diagnosing the severity and reason for a failure in a large system is challenging. The diagnosis usually requires domain and site-reliability knowledge as well as automated observability support [57]. Not all companies have such resources.

3.3 Evaluating Failure in Component-Based Multi-Tier Software Stacks

3.3.1 Case Studies

To provide some evidence for the reliability claims in Section 3.2, this section explores and compares the reliability properties of microservices and component-based web stacks. Specifically, a comparison is made of two 4-tier Jupyter web stacks and a 4-tier WordPress application as follows. (1) JAPyL consists of a Jupyter notebook, an Apache web server, a Jupyter Python [113] kernel and Linux (2) JPyL is an identical stack but the Apache component has been replaced with loosely coupled Flask microservices [128]. (3) WordPress consists of Linux, Apache, MySQL & PHP (LAMP).

JAPyL A common way to make Jupyter Notebooks web accessible is to use the JAPyL web stack (Figure 3.2) and embed the Notebook into a webpage or site with an Apache Web Server as a Reverse Proxy [126]. Here, Apache is usually interposed between the client and the Jupyter Server, taking requests from clients and forwarding them to Jupyter as discussed in Section 2.1.2.

JPyL (Loosely Coupled Microservices) Migrating from a monolithic architecture towards a full microservices architecture is a gradual process in production environments. It is common for developers to initially integrate one or more microservice tiers that function alongside monolithic components/tiers as discussed in Section 2.4.2. This hybrid technique is referred to as the microlith architecture [143].

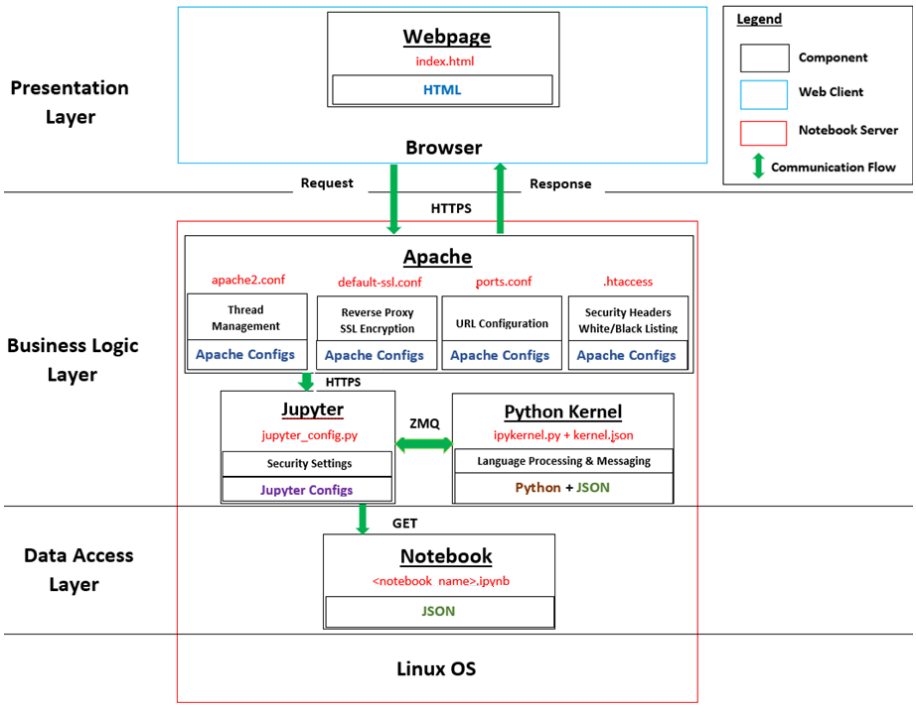

Figure 3.2: JAPyL 4-Tier Architecture

Jupyter, Python & Linux (JPyL) is a 4-tier web stack with a microlith architecture. Here the Apache component is replaced and its core functionalities provided by seven loosely coupled microservices shown in Figure 3.3. Appendix C.1 provides an example of the reverse proxy and port configuration functionalities running as a single microservice.

WordPress (Component-Based) WordPress is an open source web development and Content Management System (CMS) accounting for a significant proportion of online sites. One of the main reasons for the popularity of WordPress is its wide range of plugins that provide additional functionality [111]. As a standalone application WordPress has a multi-tier architecture with core CMS components that communicate with a MySQL database. It is usually hosted with Apache web server shown in Figure 3.4.

3.3.2 Experiment Design

The experiments run the web stacks against simulated workloads, and inject failures at specified times in a specified component/microservice. Requests are generated for each application using wrk 1.2 [46] for 60 seconds with a simulated 100 concurrent users. A five second delay failure (Appendix C.2) is introduced after 43s in the reverse proxies of the web stacks. To minimise variability the benchmarks are executed 3 times with a hot start, and the median value is reported.

The case study web stacks execute on a typical server, i.e. one node of the Glasgow Parallelism Group (GPG) cluster. These nodes, at the University of Glasgow, have a 16 core Intel

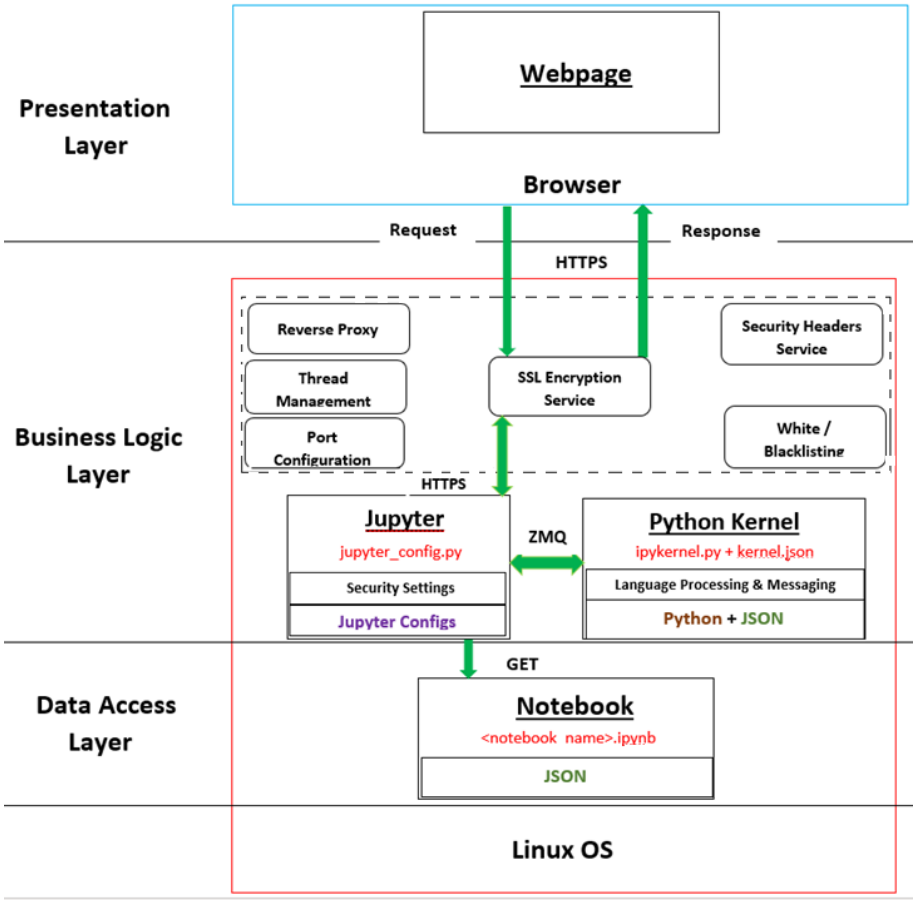

Figure 3.3: JPyL 4-Tier Architecture

server with 2TB of RAM running Ubuntu 18.04. JAPyL uses Jupyter Server 6.1 and Apache 2.4, JPyL uses Jupyter Server 6.1, Python 3.6 and Flask 1.1.2. WordPress v5.7.2, PHP 7.2 and MySQL 5.7. The code for all applications are available[1].

3.3.3 Catastrophic Failure in Component-Based JAPyL & WordPress

Catastrophic failure is a major challenge for conventional web stacks. JAPyL & WordPress are no exceptions. A failure is introduced after 43s. In Apache the failure is in the reverse proxy *mod_proxy.c* source file. Similarly, in JPyL, a failure is introduced in the Flask reverse proxy service shown in Appendix C.2.

Figures 3.5 & 3.6 show that after 43s, the Apache server for both JAPyL & WordPress loses throughput critically. By 50s, the Apache server is no longer functioning. Both stacks have suffered catastrophic failure. Data cannot be displayed via the webpage component and HTTP requests cannot be accepted.

[1] `https://bitbucket.org/latent12/microproject/src/master/microcomp/`

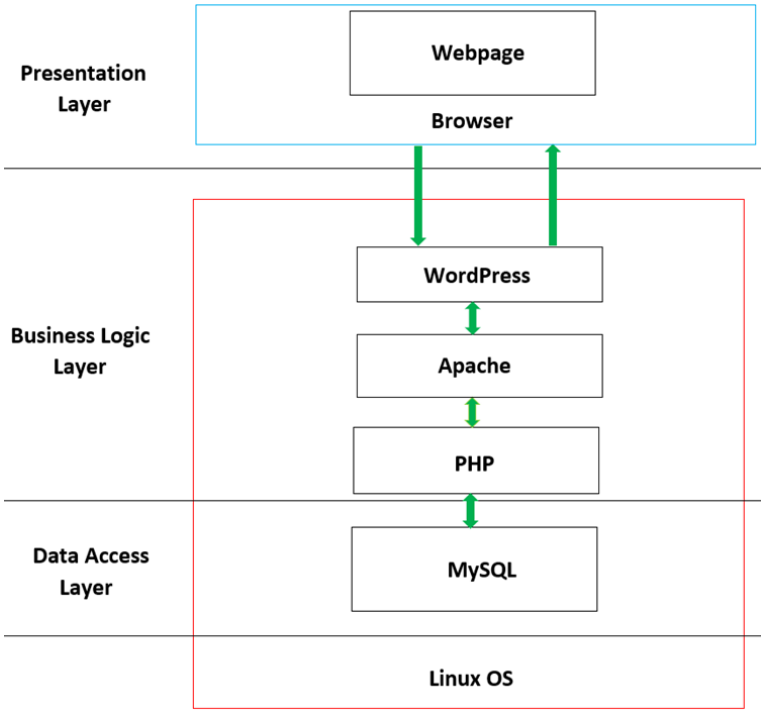

Figure 3.4: Component Based WordPress Architecture

3.3.4 Failure in Microservices-Based JPyL

To illustrate that microservices provide better reliability than components in multi-tier software stacks, failure in the component-based JAPyL is compared with failure in micro-service based JPyL. Again, a five second delay failure is introduced after 43s in the reverse proxies of both web stacks shown in Appendix C.2.

Figure 3.7 shows that Apache fails and JAPyL is unable to display any content. In contrast, in JPyL, the Flask reverse proxy and port microservices are independent and so able to switch to the new port (using portToggle()) and continue serving requests. However, there is some performance degradation. JPyL plots throughput (Request KB/s) against time. Throughput falls

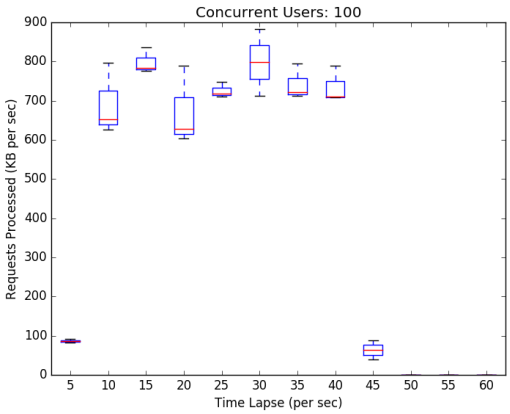

Figure 3.5: JAPyL Apache Failure at 43s.

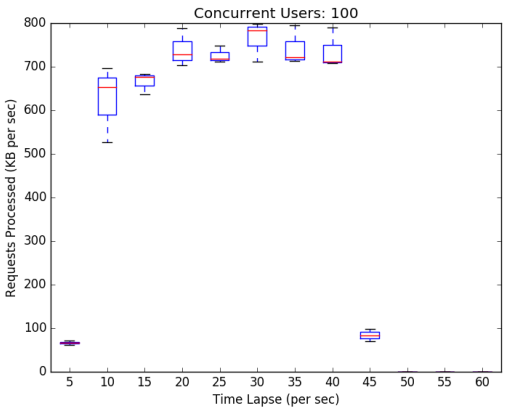

Figure 3.6: WordPress Failure at 43s.

from a peak of around 700KB/s to 160KB/s after 45s and only reaches a level of 400KB/s after 60s.

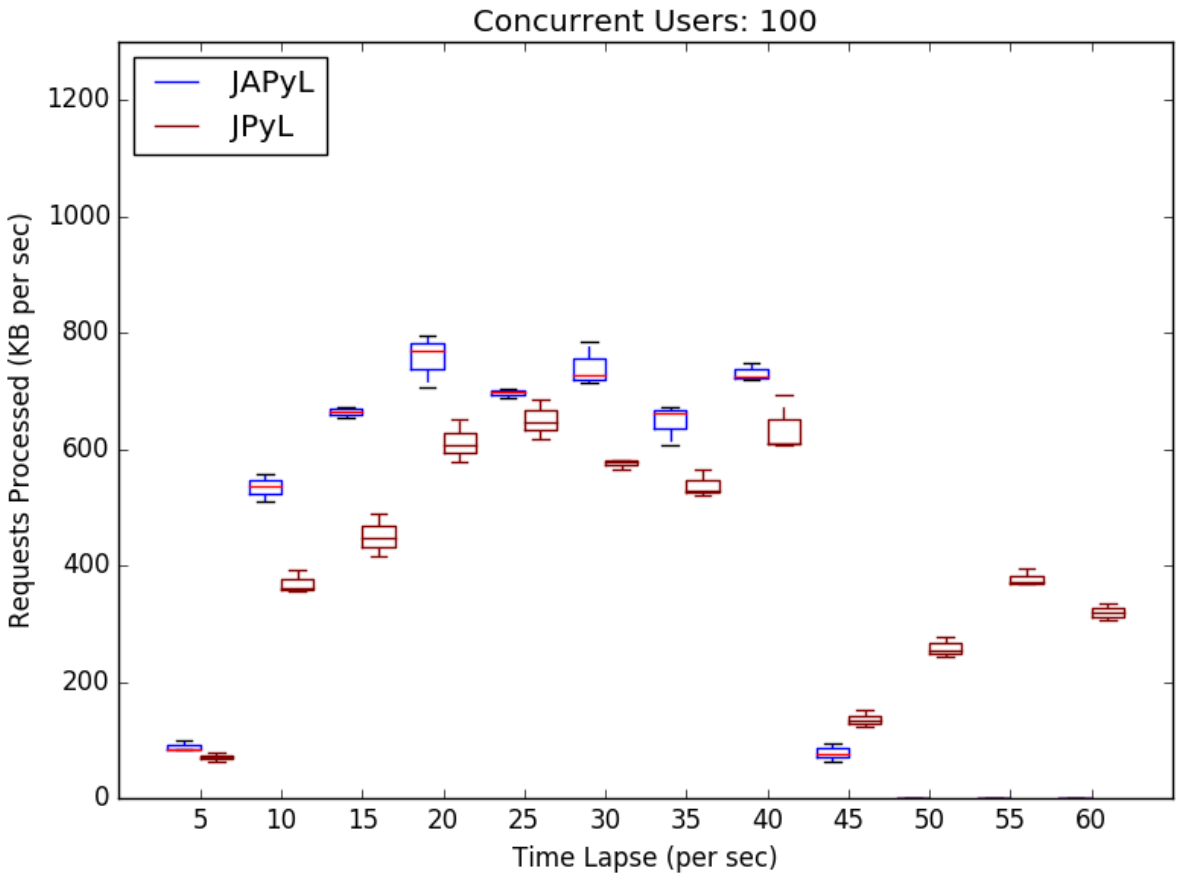

Figure 3.7: JAPyL vs JPyL Reliability Comparison

One possible explanation is that in order for the stack to efficiently handle increasing volumes of requests from users through a particular port, the data requested is initially retrieved from the server. These initial requests normally increase the workload of the server and cause performance overhead. The requested data is soon stored in cache, allowing for faster retrieval [121].

However, cache is volatile and even a short interruption to the server empties the cache [121]. Thus, during failover in JPyL, this small interruption causes the cache to be emptied. When a new port for receiving requests is assigned, the entire data retrieval and caching process has to be restarted.

3.4 Classifying the Reliability of Microservices

These experiments show that microservices do indeed improve the reliability of multi-tier software stacks. Partial failure and improved reliability are to be expected in individual microser-

vices. They are loosely coupled to other microservices, with low dependency. Communication with other services is usually via infrequent remote API calls. Many individual microservices express computations at a high level of abstraction, and provide their own built-in runtimes, functionalities and data stores [133].

This chapter now investigates the reliability properties of microservices. It makes the following contributions [123]:

(1) The *reliability* (minor/critical) classification is combined with the established classifications of dependence (individual/chained) and state (stateful/stateless). If a minor microservice fails the application continues to function, although performance or functionality may be reduced. If a critical microservice fails, the application fails catastrophically. Combining reliability with state and dependence defines a new three dimensional space: the Microservices Dependency State Reliability (MDSR) classification. As microservice chains are necessarily critical [60], only six of the possible eight points in the space are valid. A prototype static analyser is outlined that can identify all six MDSR classes. Applying the tool to 30 microservices from seven small Flask web applications reveals interesting statistics, e.g. the majority of services are chained (70%), and critical (77%) (Section 3.8).

(2) Using three web applications microservices that exemplify each point in MDSR is highlighted. The web applications are: (1) *Hipster-Shop*, a Google demo application; (2) *JPyL*, a Jupyter Notebook/Flask web stack; and (3) *WordPress*, a content management system (Section 3.5).

(3) It is shown that each of the MDSR critical case study microservices exhibits a known bad smell [149]. Likewise in each minor MDSR class the case study microservices follow a design pattern [149]. The prototype static analyser can identify three of six patterns/bad smells in Flask web applications. The analysis offers the opportunity to focus reliability engineering efforts early in the development cycle. That is, it is proposed that *static* MDSR analysis act as a complement to *dynamic* Service Dependency Graph (SDG) analysis [89] (Section 3.8).

(4) The reliability implications of different MDSR microservice classes is explored by running the three web applications against a simple process level fault injector. Specifically it is shown: (1) *All* applications fail catastrophically if a critical microservice fails. (2) Applications survive the failure of a minor microservice, and successive failures of minor microservices. (3) The failure of any chain of microservices in JPyL & Hipster is catastrophic. (4) Individual microservices do not necessarily have minor reliability implications (Section 3.9).

3.5 Microservices Classification Case Studies

To illustrate the new findings in this chapter, three realistic microservice web applications are utilised. *Hipster-Shop* is a popular Google microservices demonstration web application; *JPyL* has been updated with different classes of microservices; *WordPress* has been updated to facilitate integration of microservices plugins. These illustrate different aspects of real world microservice web applications, e.g. Hipster-Shop implements microservices in different languages, and both JPyL and WordPress combine components and microservices.

3.5.1 Hipster-Shop

Key attractions of microservices are decentralized and polyglot development. Here each service can be separately developed utilising some suitable programming language and tools, and this promotes agile development [171]. Indeed many developers cite this as a reason for their preference for microservices [66].

The Hipster-Shop case study illustrates polyglot microservice development with services developed in Python, Go and Java with communication via gRPC remote procedure calls. The Hipster-Shop is an e-commerce application with ten microservices (Figure 3.8) used by Google to demonstrate tools like Kubernetes Engine [47]. Users can perform activities like viewing products, adding items to cart and making purchases [2].

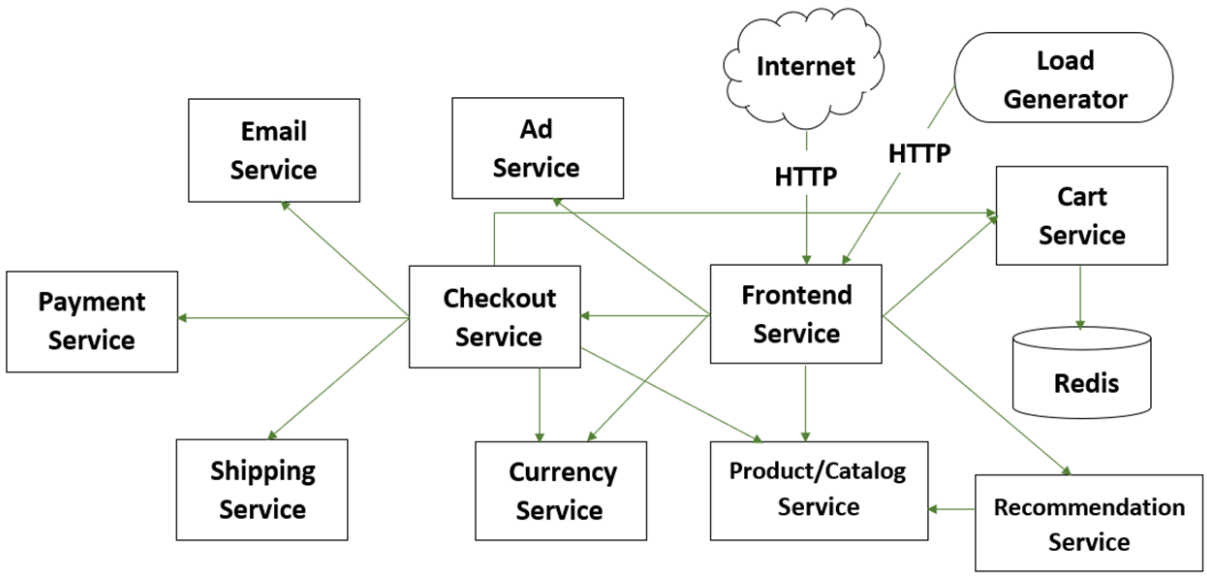

Figure 3.8: Hipster-Shop Architecture [3]

[2]Source: `https://github.com/GoogleCloudPlatform/microservices-demo`

3.5.2 JPyL ver2.0

This version of JPyL has been upgraded to consist of different classes of microservices. Some are supported by a data store, e.g. current geolocation and IP address information are accessed via the userdata microservice that extracts the data from a MySQL database [123].

Data is displayed on the webpage via *the chained reverse proxy and port configuration* microservices on port 10125. Crucially for reliability a backup URL port can be initiated via a redirect if the original port service is interrupted. Users will still be able to access the content and are automatically redirected to a different port [123]. Appendix C.3 provides a code snippet of this new reverse proxy and port configuration chain implementation in JPyL.

Each service is handled by a specific Flask microservice or set of microservices[4]. For example, security headers are processed by the Python Talisman microservice. It automatically enforces the use of HTTP security headers by configuring the behaviors within the browser and server once the web app is running.

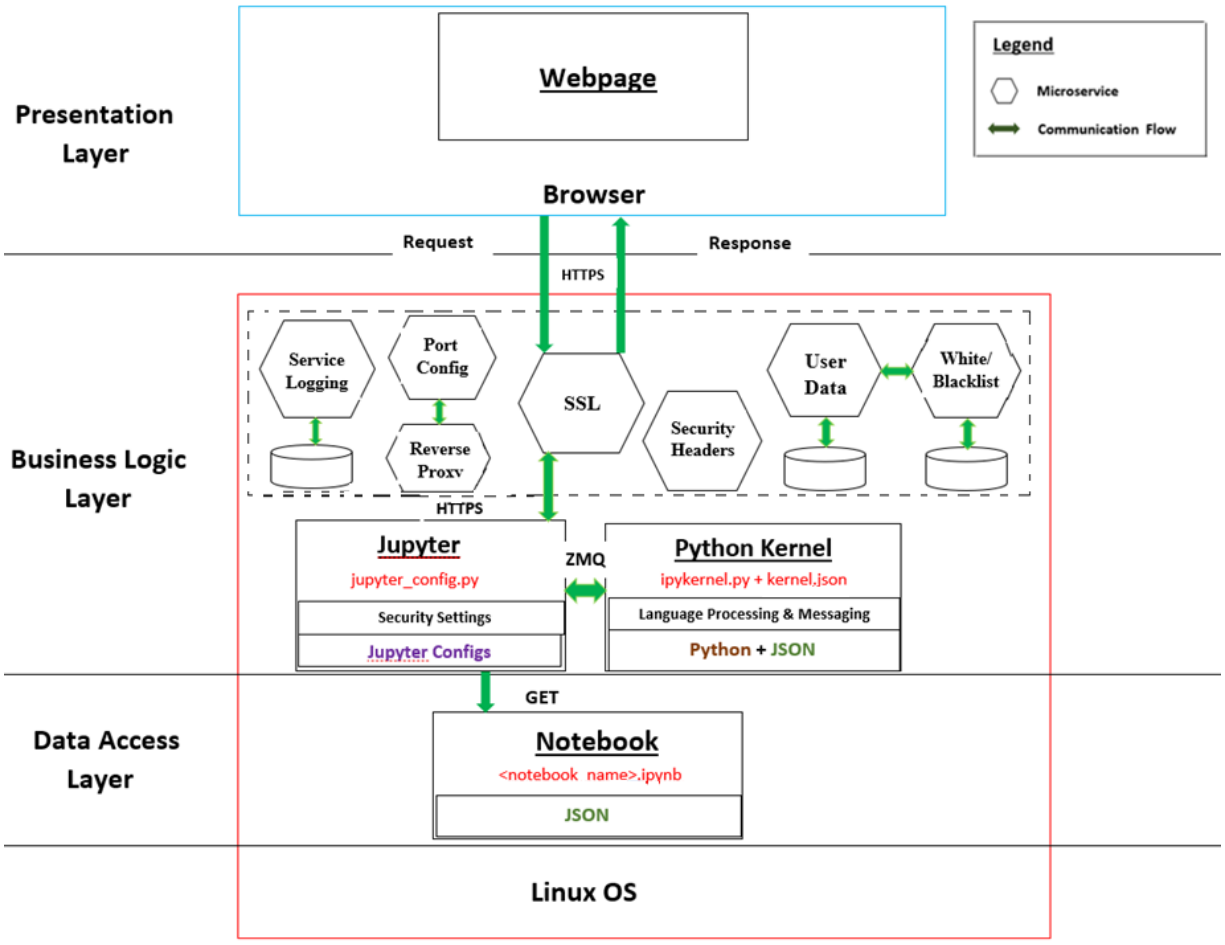

Figure 3.9: JPyL with Different Classes of Microservices

[4] https://bitbucket.org/latent12/microproject/src/master/jpyl/

3.5.3 WordPress ver2.0

Microservices can be integrated with WordPress to provide plugins that support additional functionalities. For example to provide facilities to post comments, allow subscription memberships, to search indexes, or to provide data analytics [18].

In this upgraded version, The application integrates microservices endpoints that allow users to post comments on a blog[5]. The service utilises WordPress HTTP REST. The main function of this service is to facilitate communication between other microservices and the components shown in Figure 3.10.

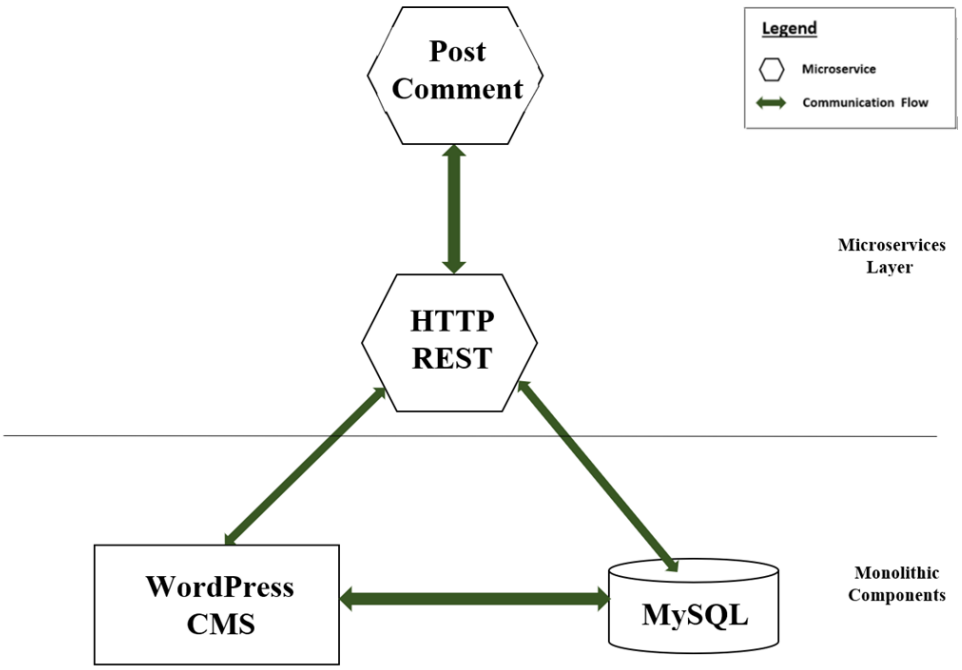

Figure 3.10: WordPress Microservices Application

3.6 Microservices Properties

A limitation of [60] and the Lewis and Fowler design principles is that they consider only the microservices property of dependence [123]. There are other types of properties like state. Microservices are classified by their properties and some key properties are outlined below and summarised in Table 3.1.

3.6.1 Dependence

This property classifies a microservice by how tightly coupled it is with other microservices [45, 60, 89]. Coupling is the degree of dependence between software components like microservices,

[5] https://bitbucket.org/latent12/microproject/src/master/wordpress/

Table 3.1: Microservices Classification Criteria

Classification	Properties	Description
Dependence	Individual	Loosely Coupled. Constant communication with other microservices not required.
	Chained	Tightly Coupled. Constant communication with other microservices required.
State	Stateless	No data store. Does not maintain state.
	Stateful	Utilises data store. Maintains state.
Reliability	Critical	Supports core functionality. Service failure means application becomes suddenly and permanently unavailable.
	Minor	Supports non-essential functionality. Application continues to function despite service failure. Degradation in performance or graceful failure over a period of time.

and there are different types like content, data and control coupling [107]. Software architects seek loose coupling, and this is often achieved for microservices through data coupling discussed in Section 2.3.2.

Individual microservices are loosely coupled to other microservices and communication with other microservices is typically via infrequent remote API calls. Many individual microservices express computations at a high level of abstraction, and provide their own built-in runtimes, functionalities and data stores [133]. Examples for JPyL include the SSL and Service Logging services (Figure 3.9) while Hipster-Shop includes Frontend and Adservice microservices (Figure 3.8).

In contrast *chained microservices* are tightly coupled with one or more other microservices. A chained microservice is reliant on some form of constant communication or a *chain of calls* with another service to function [60, 130]. This is often due to control coupling where the chained services must request and marshal data between themselves [107].

3.6.2 State

This property classifies a microservice by whether it preserves state between service requests. *Stateful* microservices require data storage, for example to record transactions or current actors [166]. In JPyL the Service Logging (Figure 3.9) and the UserData/White-Blacklist microservices are stateful (Figure 3.11). They log or store transactions in the application in a

MySQL database. Other microservices are *Stateless*, i.e. they maintain no session state. Such services typically accept requests, process them in a pure fashion, and respond accordingly. In JPyL the SSL microservice is stateless: it processes HTTPS requests but maintains no session data.

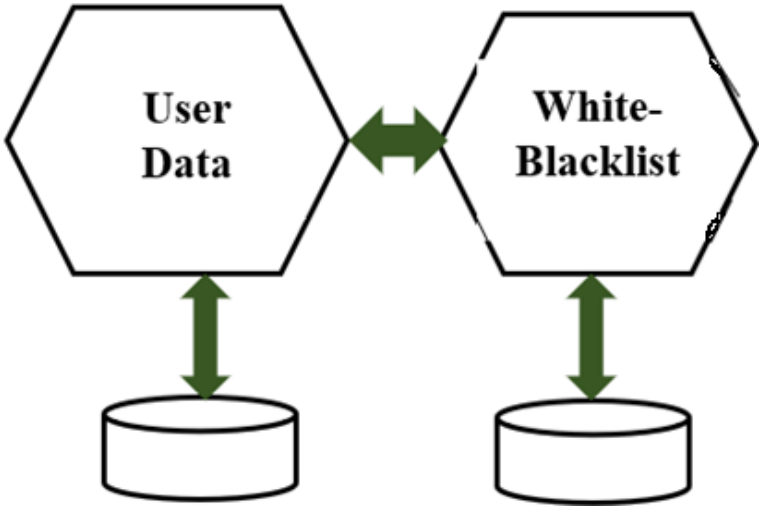

Figure 3.11: Stateful Microservices Example

3.6.3 Reliability (Critical vs Minor)

To analyse the reliability of a microservice architecture this study considers a microservice reliability property alongside the established properties of state and dependency.

Critical microservices provide core functionality to the application, and if such a service fails, the entire application fails catastrophically even if there are several instances of the microservice. In JPyL, the chained PortConfig to ReverseProxy services are critical because if the PortConfig service fails, the ReverseProxy service will not be able to determine the port to display data or access the URL backup port. As with other properties criticality is inherited within chains, so if any microservice is critical then the entire chain is critical [123].

Minor microservices provide non-essential functionality. The application continues to operate if they fail, although performance and/or functionality may be reduced. In Hipster-Shop, the Adservice microservice is minor because if it fails, the server returns a 404 status code indicating that the service is not found. However, the rest of the application continues to function normally [123].

It would also be possible to consider partial failures that eventually affect the operation of the system [29]. However, given the challenges of distinguishing between partial failures with different severities a binary minor/critical classification is adopted.

Partial failures are considered as gray failures. They can be subtle at first. However, these failures can eventually become catastrophic overtime depending on how long they are left unde-

tected or unresolved [62]. Thus, severe partial failures are classified as critical, and low severity failures are classified as minor [123].

3.6.4 Combining Microservices Properties

Microservices may have any combination of properties, e.g. individual/stateful or chained/stateless. Properties may be inherited from other chained microservices, e.g. if any microservice is stateful then the entire chain is stateful. In Hipster-Shop although both Checkout and Payment microservices are stateless, their chain with Cart Services is stateful as Cart Services is stateful [123].

3.7 Catastrophic Failure in Microservices

3.7.1 Chained Failure

Catastrophic failure is considered unacceptable in microservices implementation as it can lead to long downtimes without manual intervention [105]. In microservices, catastrophic failures are often termed Interaction Failures [170]. Common causes are incorrect coordination or communication failure between microservices, e.g. asynchronous message delivery lacking sequence control or a microservice receiving an unexpected output in its call chain. The errors may be replicated in several microservice instances [170], so even switching workload from a failed instance doesn't help as the new instance fails in the same way.

Chained microservices are especially prone to interaction faults because they violate the *Single Responsibility Principle (SRP)* and lead to brittle architectures [60]. Moreover adding more microservices to the chain increases coupling [39] and the likelihood of catastrophic failure. If one service in the chain fails, there will be a cascade of failures of all services in the chain [60, 158].

3.7.2 Patterns and Bad Smells

Some design patterns capture reusable solutions to common microservice design challenges [149]. For example the *Database-Per-Service* pattern prevents tight coupling by ensuring that multiple microservices are not dependent on a single data store. Instead, each service accesses its own private store [150], eliminating the single data store as a single point of failure (SPOF). Some of the microservices patterns utilised in our evaluation are summarised in Table 3.2

While design patterns like Database-Per-Service help, they are not universal solutions. For example a single atomic operation often spans multiple microservices, and here additional techniques are required to ensure consistency across the data stores [131].

Table 3.2: Microservices Patterns [150]

Pattern	Description
Database-Per-Design	Microservice accesses its own private data store.
API-Gateway	Microservice communication occurs through an API gateway.
Single Responsibility Principle	Microservice performs a single functionality

Likewise microservice bad smells identify common designs that may cause issues [149]. Indeed [60] and the Fowler and Lewis design principles consider *all chained microservices* as bad smells and prone to catastrophic failure.

The code snippet in Appendix C.3 shows an instance where the Reverse Proxy microservice is chained to the Port Config microservice in JPyL to receive data. If the Port Config microservice is interrupted or fails, the Reverse Proxy microservice will no longer receive the data it needs to function. This is an instance of the *Inappropriate Service Intimacy* bad smell [150].

Some of the microservices bad smells found in the evaluation are summarised in Table 3.3. Microservices Greedy, Shared Persistency and Cyclic Dependency are listed in [149], Chained Services is mentioned by [60] and SRP Violation is a well-known microservice bad smell.

Table 3.3: Microservices Bad Smells

Bad Smell	Description
SRP Violation	Microservice performs more than one functionality. Reason: Microservice becomes more critical. Increases the probability of catastrophic failure in the application.
Microservices Greedy	Microservices created for every feature in an application. Reason: More microservices could lead to more points of failure
Shared Persistency	Different microservices access the same data storage. Reason: Single Point of Failure (SPOF)
Chained Services	Microservices that depend on communication or data marshalling from other microservices in order to function. Reason: Tight Coupling
Cyclic Dependency	Where there are cycles in the call graph, e.g. A to B, B to C and C to A. A subset of Chained Services. Reason: Too much dependency.

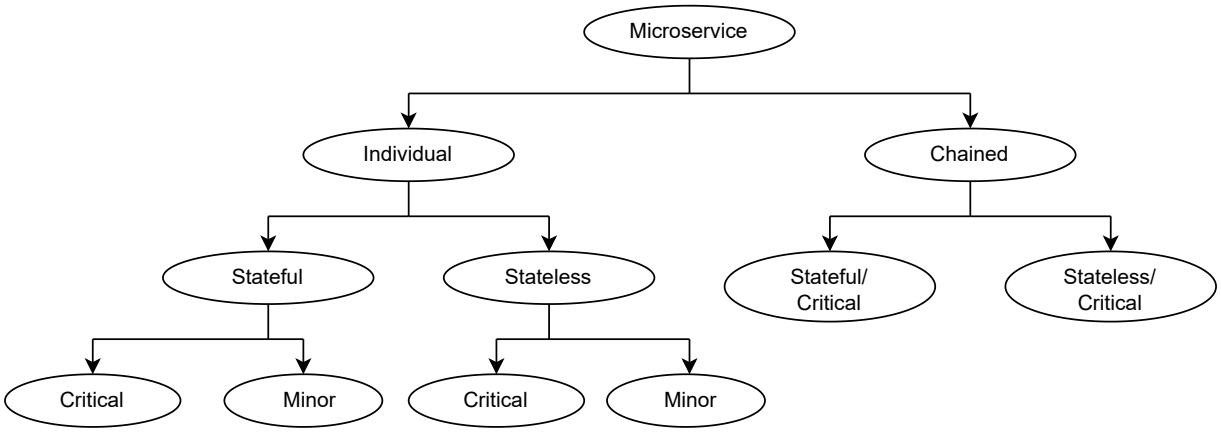

Figure 3.12: MDSR Classification Tree.

3.8 Microservices Dependency State Reliability (MDSR) Classification

3.8.1 Classification Schema

Combining these properties defines a three-dimensional space: a new Microservices Dependency State Reliability (MDSR) Classification Tree shown in Figure 3.12. The notion is that a microservice can have any combination of properties e.g. individual/stateful/minor or chained/stateless/critical [123]. It can also be represented in tabular form as in Tables 3.4 and 3.5.

Table 3.4: MDSR Pattern Classification.

	Individual				Chained	
	Stateful		Stateless		Stateful	Stateless
	Critical	Minor	Critical	Minor	Critical	Critical
Microservices		**JPyL** Service Logging		**Hipster** Adservice **JPyL** Security Headers	**JPyL** User Data, Black Listing	**Hipster** Recommend, Product Catalog
Failure Impact		404 Service Not Found		404 Service Not Found	404 Service Not Found	404 Service Not Found
Pattern		Database Per Service		Single Responsibility Principle (SRP)	Database Per Service	API Gateway

In MDSR chained microservices are necessarily critical as argued in [60], and confirmed in our evaluation (Section 3.9) even for chains that attempt to recover reliability using microservice patterns. As examples a Database-Per-Service pattern is implemented for the chained/stateful

Table 3.5: MDSR Bad Smell Classification.

	Individual				Chained	
	Stateful		Stateless		Stateful	Stateless
	Critical	Minor	Critical	Minor	Critical	Critical
Microservices	**Hipster** Frontend		**JPyL** SSL		**Hipster** Payment, Checkout, Cart **WordPress** Comments, HTTP REST	**Hipster** Shipping, Checkout **JPyL** PortConfig, Reverse Proxy
Failure Impact	500 Internal Server Error		ERR_SSL Protocol Failure Error		Database Connection Error	500 Internal Server Error
Bad Smells	SRP Violation		Microservices Greedy		Chained, Shared Persistency	Chained, Cyclic Dependency

UserData & White/Black Listing service and an API Gateway pattern for the chained/stateless/ Product Catalog & Recommended service. In both cases the application fails catastrophically despite reporting only a "404 Service Not Found" error [123].

The fourth rows of Tables 3.4 and 3.5 show example microservices from the case study applications for each of the 6 MDSR classes. For example the individual/stateful/minor exemplar is JPyL's Service Logging microservice. The fifth rows of the tables show the error reported if the service fails [123].

3.8.2 Semi-Automatic MDSR Classification

Static analysis of a set of microservices can automatically propose MDSR classifications for many microservices in an application. This principle is demonstrated with a prototype analyser that classifies all Python/Flask microservices in a source project [6]. The analyser tokenises the Python/Flask code, and identifies properties using keyword matches. As examples, the presence of keywords like "SQL" or "JSON" classifies a service as stateful; the presence of "request", "requests", "requests.get", "get" Flask keywords, or use of the "POST" or "GET" methods classifies a service as chained as they indicate a service is pushing data or requesting information from other microservices. Reliability is determined by the type of type of pattern or bad smell detected discussed in Sections 3.8.5 & 3.8.6. Figure 3.13 shows a screenshot of the tool's output for JPyL.

The analyser performs an initial assessment of all the microservices in the application. If

[6] https://bitbucket.org/latent12/microproject/src/master/analyser/

```
MDSR Classification Properties

('ssl.py', '[Individual', 'Stateless', 'N/A]', 'Reason: No Assignment')
('listing.py', '[Chained', 'Stateful', 'Critical]', 'Bad Smell: Chained')
('userinfo.py', '[Chained', 'Stateful', 'Critical]', 'Bad Smell: Chained')
('proxy.py', '[Chained', 'Stateless', 'Critical]', 'Bad Smell: Chained')
('logs.py', '[Individual', 'Stateful', 'N/A]', 'Reason: No Assignment')
('portconfig.py', '[Chained', 'Stateless', 'Critical]', 'Bad Smell: Chained')
('sechead.py', '[Individual', 'Stateless', 'N/A]', 'Reason: No Assignment')


Proceeding To Analyse Microservices For Patterns/Bad Smells



MDSR Patterns/Bad Smells

('ssl.py', '[Individual', 'Stateless', 'Critical]', 'Failure: ERR_SSL')
('listing.py', '[Chained', 'Stateful', 'Critical]', 'Bad Smell: Chained')
('userinfo.py', '[Chained', 'Stateful', 'Critical]', 'Bad Smell: Chained')
('proxy.py', '[Chained', 'Stateless', 'Critical]', 'Bad Smell: Chained')
('logs.py', '[Individual', 'Stateful', 'Minor]', 'Pattern: Database-Per-Service')
('portconfig.py', '[Chained', 'Stateless', 'Critical]', 'Bad Smell: Chained')
('sechead.py', '[Individual', 'Stateless', 'Minor]', 'Failure: 404')
```

Figure 3.13: JPyL Output From MDSR Analyser

chained microservices are detected, it is automatically flagged as critical as this considered a bad smell by default. Individual microservices are not assigned a reliability property during the initial assessment. A second assessment is then performed on all microservices. At this stage, individual microservices are now assigned a reliability property. Two decision making parameters, i.e Patterns/Bad Smells Analysis and Error Handling Codes, are then used to determine and provide further information on the reasons why the reliability property is assigned. Patterns/Bad Smells Analysis takes precedence [123].

The prototype analyser identifies all six MDSR classes. The analyser may, however, propose an incorrect classification, for example a stateless service may be incorrectly classified as stateful if a keyword like "SQL" appears in a comment. Similarly, a stateful microservice could be classified as stateless if it uses a persistent store that is not included in the current set of keywords [123].

Despite these limitations the analyser is effective in classifying microservices. For example Tables 3.4 & 3.5 show how it correctly classifies all of the JPyL microservices. The analyser has been applied to a total of 30 microservices in a further six small Flask web application projects[7]. Manual inspection of three of the projects validates the properties identified by the analyser. As a further example the analyser output for the IBM worklog application is shown in Figure 3.14,

[7]GitHub Links: https://github.com/IBM/worklog/tree/master/app, https://github.com/IBM/Flask-microservice, https://github.com/bakrianoo/Flask-elastic-microservice, https://github.com/airavata/Blitzkrieg, https://github.com/michaellitherland/Flask-microservice-demo, https://github.com/umermansoor/microservices

```
MDSR Classification Properties

('create_user_service.py', '[Individual', 'Stateful', 'N/A]', 'Reason: No Assignment')
('user.py', '[Chained', 'Stateful', 'Critical]', 'Bad Smell: Chained')
('login.py', '[Chained', 'Stateful', 'Critical]', 'Bad Smell: Chained')
('get_worklog_data_service.py', '[Chained', 'Stateful', 'Critical]', 'Bad Smell: Chained')
('reset_password_service.py', '[Chained', 'Stateful', 'Critical]', 'Bad Smell: Chained')
('login_service.py', '[Chained', 'Stateful', 'Critical]', 'Bad Smell: Chained')


Proceeding To Analyse Microservices For Patterns/Bad Smells



MDSR Patterns/Bad Smells

('create_user_service.py', '[Individual', 'Stateful', 'Minor]', 'Pattern: Database-Per-Service')
('user.py', '[Chained', 'Stateful', 'Critical]', 'Bad Smell: Chained')
('login.py', '[Chained', 'Stateful', 'Critical]', 'Bad Smell: Chained')
('get_worklog_data_service.py', '[Chained', 'Stateful', 'Critical]', 'Bad Smell: Chained')
('reset_password_service.py', '[Chained', 'Stateful', 'Critical]', 'Bad Smell: Chained')
('login_service.py', '[Chained', 'Stateful', 'Critical]', 'Bad Smell: Chained')
```

Figure 3.14: IBM Worklogs Output From MDSR Analyser

and corresponds to the use case diagram provided by IBM[8].

Table 3.6 shows the number and percentage of each MDSR class of microservice that the analyser detects in the seven web applications, and key observations are as follows. The majority of services are chained (70%), and most are stateful (73%). 50% of services are from a single MDSR classification, i.e. chained/stateful/critical. Perhaps most startling is that 77% of services are critical. It is possible to speculate that this reflects that the designers of these small web applications have not designed them to be reliable [123].

3.8.3 MDSR Patterns & Bad Smells Analysis

MDSR analysis provides information about the expected reliability of microservices and chains of microservices in an architecture. In general reliability engineering should focus on the 77% of critical microservices identified by the analysis in Table 3.6. More specifically the analysis can help identify design patterns and bad smells in the architecture. To illustrate, the sixth row of the Patterns and Bad Smell tables (Tables 3.4 & 3.5) identify the microservice pattern or bad smell associated with each point in the classification space. Of the patterns and bad smells enumerated in [149] (and summarised in Tables 3.2 & 3.3) the case studies exhibit four out of eight patterns and three out of eleven bad smells.

Static analysis enables the early identification of patterns and bad smells, allowing developers to anticipate the types of failures, and their likely impact. Potentially this information allows developers to troubleshoot problems faster and prevent long application downtimes [123].

[8] https://github.com/IBM/worklog/blob/master/designs/use_case_diagram.png

Table 3.6: MDSR Classifications of 7 small Flask Applications containing 30 Microservices

Classification	No. Services	%
Individual	9	30
Chained	21	70
Stateful	22	73
Stateless	8	27
Critical	23	77
Minor	7	23
Individual/Stateful	7	23
Individual/Stateless	2	7
Chained/Stateful	15	50
Chained/Stateless	6	20
Individual/Stateful/**Critical**	2	7
Individual/Stateless/**Critical**	1	3
Chained/Stateful/**Critical**	15	50
Chained/Stateless/**Critical**	6	20
Individual/Stateful/**Minor**	5	17
Individual/Stateless/**Minor**	1	3

3.8.4 MDSR Pattern Instances & Implications

In the MDSR Patterns table (Table 3.4), the sixth row identifies the microservice pattern exhibited by the case study example microservice, or microservice chain. In our case study applications, individual/stateful microservices and individual/stateless microservices have only minor reliability implications if they implement a pattern as shown in Table 3.4. For example JPyL Service Logging is individual/stateful/minor and implements the Database-Per-Service pattern.

3.8.5 MDSR Bad Smells Instances & Implications

In the MDSR Bad Smells table (Table 3.5), the sixth row identifies the microservice bad smell exhibited by the case study example microservice, or microservice chain. Considering the Patterns and Bad Smells tables together (Tables 3.4 and 3.5) it is shown that the example case study microservices at each point in the MDSR classification exhibit either a design pattern or a bad smell. This is expected as microservice best practice applies patterns, while bad smells indicates places where design principles have not, or cannot be applied [60].

Bad smells identified by MDSR can be considered for refactoring to improve reliability. That is, most critical microservices are associated with known bad smells as shown in Table 3.5. For example the individual/stateless/critical SSL microservice in JPyL is an instance of Microservices Greedy, where there is a proliferation of microservices. Of course SSL need not be implemented as a microservice [123].

A key element of MDSR is that chained microservices remain critical. For example, the chained/stateful/critical UserData & White/Black Listing microservices implement the Database-

Per-Service pattern but still fail catastrophically as shown in Section 3.9.

3.8.6 Semi-Automatic Pattern/Bad Smell Detection

The prototype MDSR analyser can detect the Database-Per-Service pattern, and both chained microservices and Shared Persistency bad smells. Shared Persistency is detected by determining whether any microservices share a data store. Currently the user must provide the analyser with the names of the data stores used in the application, e.g. jpyl_micro in JPyL. An enhanced analyser could parse the Flask code and extract the data store names from connection statements. The analyser counts the number of times each data store name appears in the microservices in the given directory. If the count is greater than one, the microservices have a shared persistency bad smell. Microservices with a unique persistent store name implement a Database-Per-Service pattern [123].

There are some bad smells that the analyser is not able to detect. Some of these, like Cyclic Dependency could be detected dynamically, perhaps using SDGs to examine the connection between services and the rate of communication [89, 109]. Other bad smells likely require human analysis, like Microservices Greedy and SRP violation.

The analyser also inspects Flask error handling codes to classify the reliability of a microservice. For example if a service returns a 404 error indicating that the service is not found the failure is considered minor. In contrast a code like 415 indicates that there is a SSL Protocol Violation, and the service is critical because even if other services are available, the application cannot accept HTTPS requests, and has failed catastrophically [123].

Table 3.7: MDSR Patterns/Bad Smells found in 7 small Flask Applications containing 30 Microservices

Patterns/Bad Smells	No. Services	%
Database-Per-Service	5	17
Shared Persistency	2	7
Chained Services	21	70
Other[9]	2	6

Table 3.7 shows the number and percentage of bad smells and patterns detected by the analyser in the seven web applications; key observations are as follows. 17% of services implement Database-Per-Service. As 70% of the services are chained (Table 3.6), they are the most common bad smells. This accords with, and provides evidence for, the claim in [60] that developers do not always implement reliability patterns [123].

3.9 Validating MDSR Reliability

3.9.1 Experiment Design

The experiment design to validate the findings of MDSR is similar to the previous experiments conducted in Section 3.3.2. However, this fault injector is implemented in Python using the Chaos Monkey Engine 1.1.0 [120] to terminate the process associated with a specific microservice at a specific time. As neither process nor termination time is selected at random, this is not a Chaos Monkey.

3.9.2 Critical Microservices Failure

Catastrophic failure is a major challenge for web applications and our case study applications are no exception. To investigate the failure of critical microservices we target the chained/stateful/critical HTTP REST microservice in WordPress, the chained/stateless/critical Port Config microservice in JPyL, the individual/stateless/critical SSL microservice in JPyL & the chained/stateful/critical Cart Service in Hipster [123].

As in Section 3.3.2, throughput is plotted (Request KB/s) against time. The red line in each box plot is the median throughput from three executions. When the fault injector kills the critical microservice at 43s the application fails almost instantly: by 50s throughput is 0KB/s as shown in Figures 3.15, 3.16, 3.17 & 3.18.

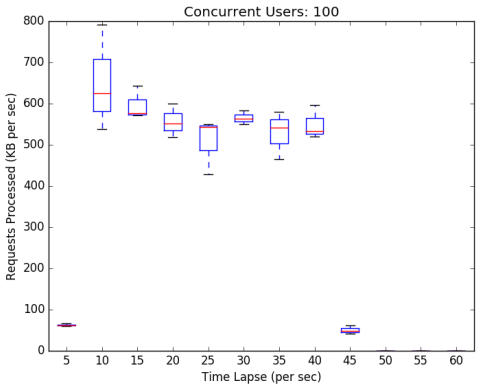

Figure 3.15: JPyL PortConfig & Reverse-Proxy Critical Failure at 43s.

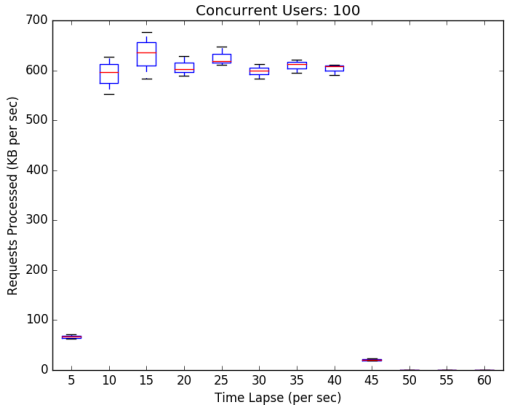

Figure 3.16: JPyL SSL Critical Failure at 43s.

This catastrophic failure is *similar to that exhibited by components in multi-tier software stacks*. For example, in WordPress, once the HTTP REST microservice failed, the application was unable to accept HTTP requests or display any data on the web page component similar to that when the Apache component was compromised for WordPress in Section 3.3.2.

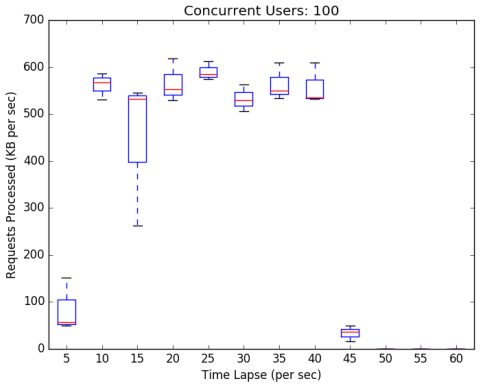

Figure 3.17: WordPress HTTP REST & Comment Critical Failure at 43s.

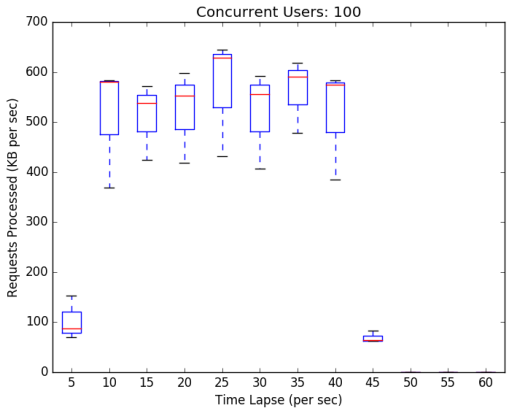

Figure 3.18: Hipster Cart & Payment Services Critical Failure at 43s.

3.9.3 Minor/Individual Microservice Failure

The first investigation of the failure of minor microservices uses an *individual* microservice. Specifically the individual/stateful/minor Service Logging microservice in JPyL is targeted. Recall that, although stateful, this microservice has a private store following the Database-per-service design pattern [123].

As before, Figure 3.19 plots JPyL throughput (Request KB/s) against time, and the service is running at around 600KB/s. Once the fault injector kills the critical microservice at 43s the application continues to serve pages, but throughput falls dramatically but briefly to around 2KB/s. By 50s the application is able to recover to a throughput of around 520KB/s.

3.9.4 Critical/Chained Microservices Failure

The next investigation involves the failure of critical/chained *chained* microservices. Specifically the chained/stateful/critical User Data & White/Black Listing microservices in JPyL and the chained/stateless/critical Product Catalog & Recommended microservices in Hipster are targeted. Both microservice chains implement patterns that aim to recover reliability (Section 3.7).

As before Figure 3.20 plots JPyL throughput (Request KB/s) against time, and the service is running at around 600KB/s. Once the fault injector kills the pair of microservices at 43s the application reports a *404 Service Not Found* error and continues to serve pages. However the throughput has fallen to around 2KB/s. That is the application is barely able to accept client requests or even load in a browser quickly. A similar failure is reported for Hipster when the Product Catalog service fails (Figure 3.21).

For realistic workloads the failure of chained/critical microservices, even with pattern implementations, has caused the applications to fail catastrophically.

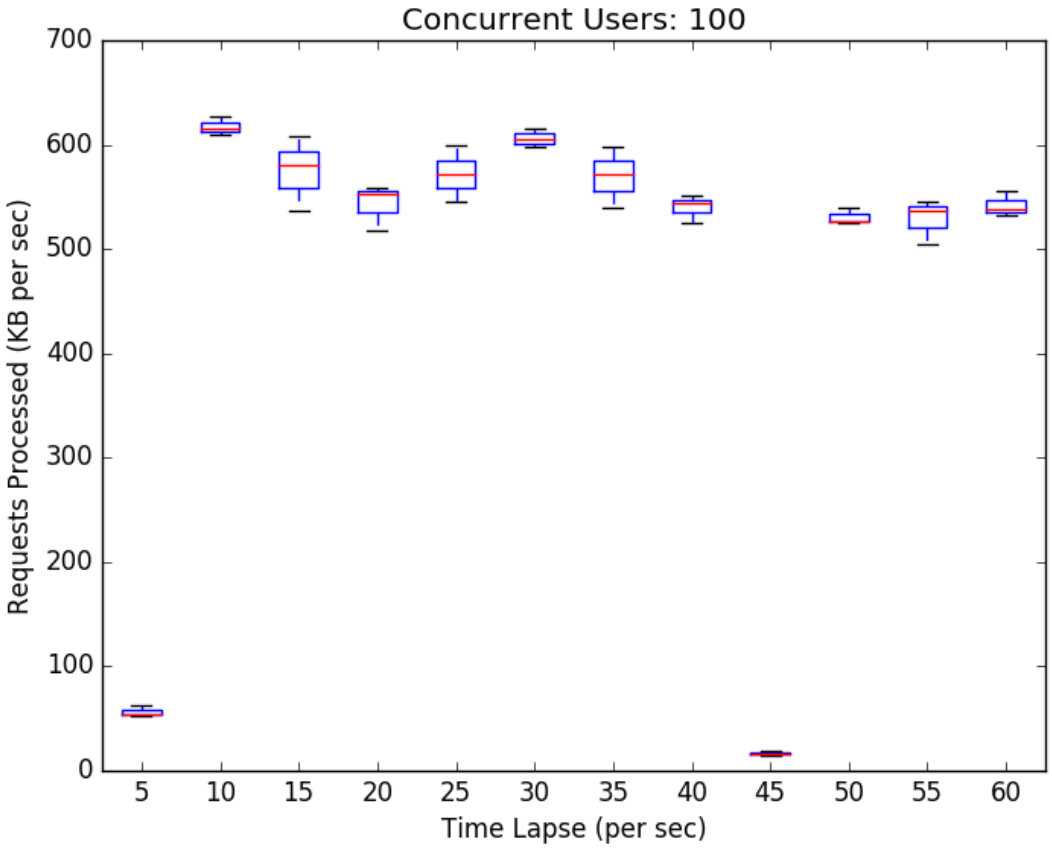

Figure 3.19: JPyL Minor/Individual Failure (Service Logging) at 43s.

3.9.5 Multiple Microservices Failure

Even if an application survives the failure of a single minor microservice, how will it cope when *multiple* microservices fail successively? To investigate the failure of multiple microservices in JPyL three microservices i.e. Service Logging, Security Headers & User Data – White/Black Listing are targeted. Specifically the fault injector kills these microservices in order at approximately 16s, 32s and 48s into the execution.

Figure 3.22 plots JPyL throughput (Request KB/s) against time, and the service is running at around 600KB/s. When the fault injector kills the *individual*/minor microservices the throughput drops briefly to around 2KB/s, but then recovers to around 600KB/s. As before, when the *chained*/critical microservice fails at 48s the application fails catastrophically.

3.10 Evaluation Summary

The key findings from the evaluation are as follows. (1) All case study applications fail catastrophically if a critical microservice fails (Figures 3.15, 3.16, 3.17 & 3.18). (2) JPyL survives the failure of an individual/minor microservice (Figure 3.19), and even the successive failure of

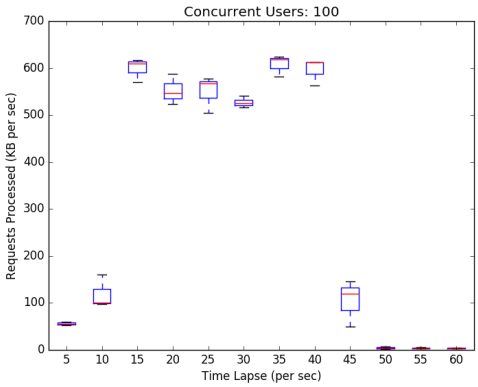

Figure 3.20: JPyL Chained (UserData & White/Blacklisting) at 43s.

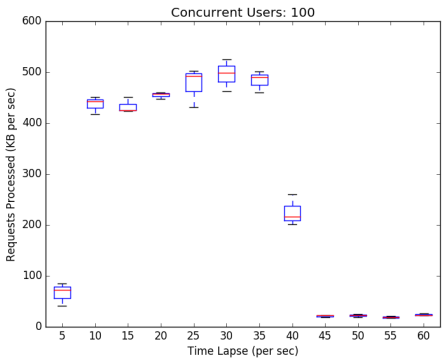

Figure 3.21: Hipster Chained (Product & Recommended) at 43s.

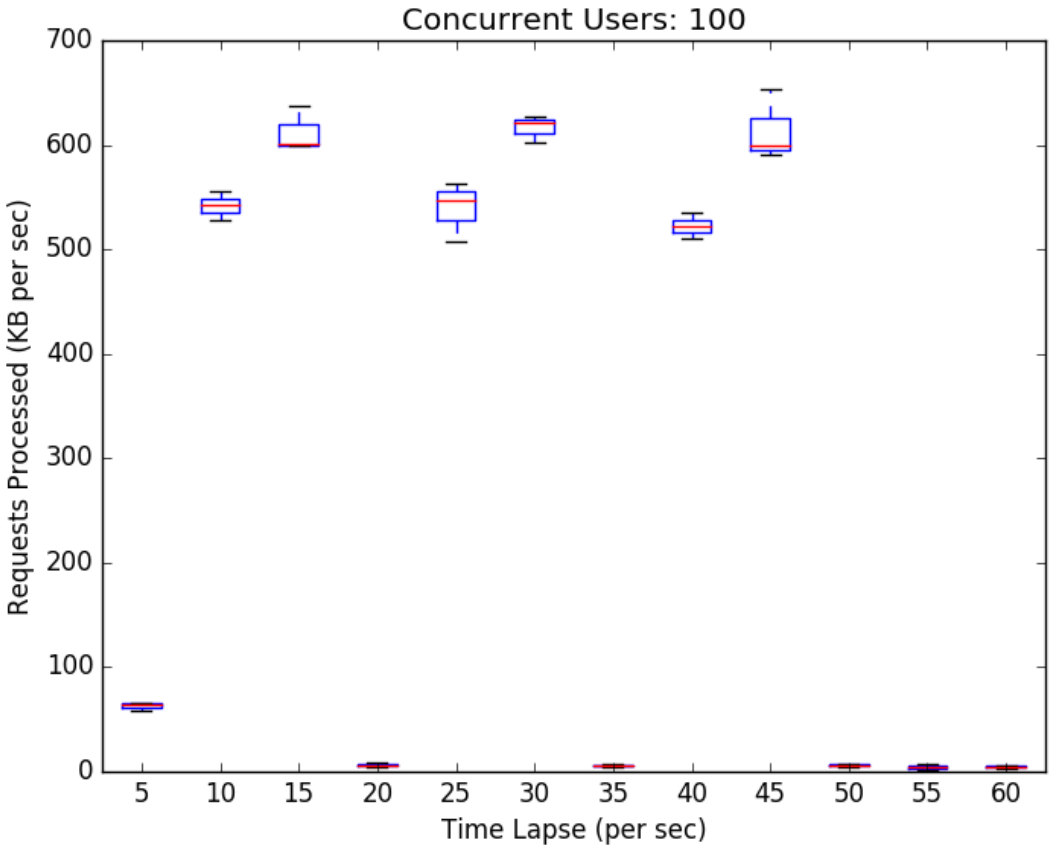

Figure 3.22: JPyL Multiple Minor Failures at 16s, 32s, 48s.

two individual/minor microservices (up to 40s in Figure 3.22) (3) The failure of any chain of microservices in JPyL & Hipster is catastrophic: throughput being dramatically reduced (by 98%) (Figures 3.20, 3.21 and after 48s in Figure 3.22). (4) Individual microservices do not necessarily have minor reliability implications, e.g. the Hipster Frontend is individual/stateful/critical and the JPyl SSL is individual/stateless/critical (Figure 3.16).

3.11 Chapter Summary

Microservices are perceived to improve interoperation in multi-tier software stacks because they provide improved reliability. This is due to the loose coupling of services. This may prevent catastrophic failure when compared to component based multi-tier software stacks that are tightly coupled (Section 3.2). Early experiments show that this notion is accurate. Component based stacks like JAPyL and WordPress fail catastrophically. In contrast, the JPyL microlith architecture prevents catastrophic failure (Section 3.3.2).

However, microservices are commonly classified based on their dependence (chained/individual) or state (stateful/stateless). A binary reliability classification is added and combined with the other classifications to define a three dimensional space: the MDSR Classification in Figure 3.12 and Section 3.8. Microservices tend to exhibit properties based on 6 MDSR classes.

A prototype static analyser is outlined that can statically identify all six classes in Flask web applications, and it is applied to seven small web applications. Analysing the applications reveal that the majority of services are chained (70%), stateful (73%) and critical (77%) (Section 3.6). Across, the seven applications it is revealed that 70% consist of the chained services that are bad smells by default while only 17% are implemented for resiliency as they consist of the Database-Per-Service pattern. It is possible that the high percentage of critical services indicates that the applications are not designed for reliability. Hence MDSR provides a framework to analyse the properties of microservices and chains of microservices in a system, identifying components to be considered for refactoring to improve reliability.

Furthermore, the reliability implications of MDSR classes is investigated by running the case study applications against a simple fault injector under realistic workloads to show the following. (1) *All* applications fail catastrophically if a critical microservice fails. (2) Applications survive the failure of individual minor microservice(s). (3) The failure of any chain of microservices in JPyL & Hipster is catastrophic. (4) Individual microservices do not necessarily have minor reliability implications.

Chapter 4

Evaluating Tierless Languages for IoT Stacks

Internet of Things (IoT) software is notoriously complex, conventionally comprising multiple tiers [88]. The developer must use multiple programming languages and ensure that the components interoperate correctly. A novel alternative is to use a single *tierless* language with a compiler that generates the code for each component and ensures their correct interoperation.

This chapter reports a systematic comparative evaluation of two tierless language technologies for IoT stacks: one for resource-rich sensor nodes (Clean with iTask), and one for resource-constrained sensor nodes (Clean with iTask and mTask). The evaluation is based on four implementations of a typical smart campus application: two tierless and two Python-based tiered.

It shows the following. (1) Tierless languages have the potential to significantly reduce the development effort for IoT systems, requiring 70% less code than the tiered implementations. Careful analysis attributes this code reduction to reduced interoperation (e.g. two embedded domain-specific languages (DSLs) and one paradigm versus seven languages and two paradigms), automatically generated distributed communication and powerful IoT programming abstractions. (2) Tierless languages have the potential to significantly improve the reliability of IoT systems, describing how Clean iTask/mTask maintains type safety, provides higher order failure management, and simplifies maintainability. (3) The first comparison of a tierless codebase is reported for resource-rich and resource-constrained sensor nodes showing that they have very similar functional structure and code sizes.

This chapter reports collaborative research [87, 88] and Section 4.1 describes my contributions to the collaboration. Section 4.2 introduces the challenges of interoperation for tiered IoT stack implementations and why tierless languages seem like a viable alternative. Section 4.3 describes the implementation of the tiered PRS & PWS. Section 4.4 describes the concept of Task Oriented Programming (TOP) and compares Clean iTask & mTask. Section 4.5 describes the implementation of the tierless CRS & CWS and shows how both the tiered and tierless IoT implementations are operationally equivalent. Section 4.6 compares and evaluates the four IoT

stack implementations. Section 4.7 summarises the chapter.

4.1 Detailed Authorship Statement

This chapter reports research undertaken in collaboration with Mart Lubbers and Pieter Koopman at the University of Nijmegen [87, 88]. The initial tiered Python/Raspberry Pi Smart Campus implementation (PRS in Section 4.3.2) was engineered by Christian Hentschel, Dejice Jacob and Jeremy Singer [59]. I developed and validated the tiered MicroPython/Wemos Smart Campus implementation (PWS in Section 4.3.3).

I helped design and conduct the experiments that compare interoperation characteristics of PRS, PWS, CRS and CWS in Section 4.6. The tierless examples in Section 4.5 and the Clean IoT implementations in Section 4.5 are engineered by Mart Lubbers and Pieter Koopman. I demonstrated the operational equivalence of PRS and PWS implementations and measured the memory residency of both in Section 4.5.2.

My PWS implementation together with Mart's CWS implementation provide the basis for the Section 4.6.2 comparison of IoT programming for resource-constrained sensor nodes with programming for resource-rich IoT nodes (as in PRS and CRS). In Sections 4.6.1, 4.6.4 & 4.6.5 while Mart Lubbers measured the CRS and CWS implementations, I measured the PRS and PWS implementations and contributed to the analysis of the result. In Section 4.6 I contributed to the analysis of the reliability of tiered and tierless IoT stacks, for example, locating an instance where type safety is lost in PRS and PWS (Section 4.6.6).

4.2 Motivation

4.2.1 Tiered IoT Stack Complexity

Tiered IoT software stacks are even more complex than tiered web stacks as discussed in Section 2.1.4. They comprise two levels of distribution where different web components interoperate with physical sensors and aggregators [87]. For example in PWS, the web components like the webpage and MongoDB in the Presentation & Application Layers have to interoperate with the Measurement Collector in the Perception Layer shown in Figure 2.6. This is facilitated through a variety of protocols like HTTP & MQTT. The components are implemented using a variety of languages like HTML, Python, JSON & PHP shown in Figures 2.12.

The tiered IoT architecture offers significant benefits like:

(1) Modularity — tiers allow a system to be structured as a set of components with clearly defined functionality. They can be implemented independently, and may be interchanged with other components that have similar functionality [90]. For example, in PWS, a different NoSQL DBMS could relatively easily be substituted for MongoDB.

(2) Abstraction — the hierarchical composition of components in the stack abstracts the view of the system as a whole. Enough detail is provided to understand the roles of each layer and how the components relate to one another [13]. Figure 2.6 illustrates the abstraction of PWS into 4 tiers.

(3) Cohesion — well-defined boundaries ensure each tier contains functionality directly related to the task of the component [79]. The tiers in PWS contain all the functionality associated with Perception, Networking, Application and Presentation Layers respectively.

However, increasing complexity creates even more heterogeneity. This presents significant challenges for interoperation as follows [87, 88]. (1) Polyglot programming increases the cognitive load for developers i.e. they have to be fluent in several different programming languages and paradigms. (2) Type safety has to be enforced and maintained across a variety of different programming languages. (3) Strict adherence to API and communication protocols must be maintained to ensure correct interoperation among components. (4) Managing different failures as different components have multiple different failure modes.

These are also evidenced in real world studies. An empirical study conducted on open source GitHub projects reveal that polyglot programming decreases project quality as it increases the chances of bug defects [71]. A study of IoT stack developers found that integrating certain architectures like microservices into tiers can decrease abstraction and that scaling tiered IoT applications to newer devices can be difficult [102].

4.2.2 Tierless Languages in IoT Software Stacks

Similar to web stacks discussed in Section 2.4.3, tierless languages may be used to address some of these challenges. This thesis claims that these languages have the potential to reduce developer effort, improve type safety, improve reliability and automatically generate communication during compilation [87, 88]. However, their use in the implementation of IoT software stacks is not as common perhaps due to IoT applications comprising microcontrollers that are resource constrained. Furthermore, tierless languages may introduce other problems like expressivity and maintainability [88].

A first ever systematic evaluation of tiered and tierless IoT architectures is reported in this chapter. Empirical evidence is provided for resource-constrained microcontrollers and resource-rich super sensors. Four implementations of a typical smart campus IoT stack are compared. Two are tiered Python/MicroPython stacks: Python, RaspberryPi Super Sensor (PRS) and Python, Wemos Super Sensor (PWS). The other two are the tierless implementations: CRS (Clean Raspberry Pi System) and CWS (Clean Wemos System).

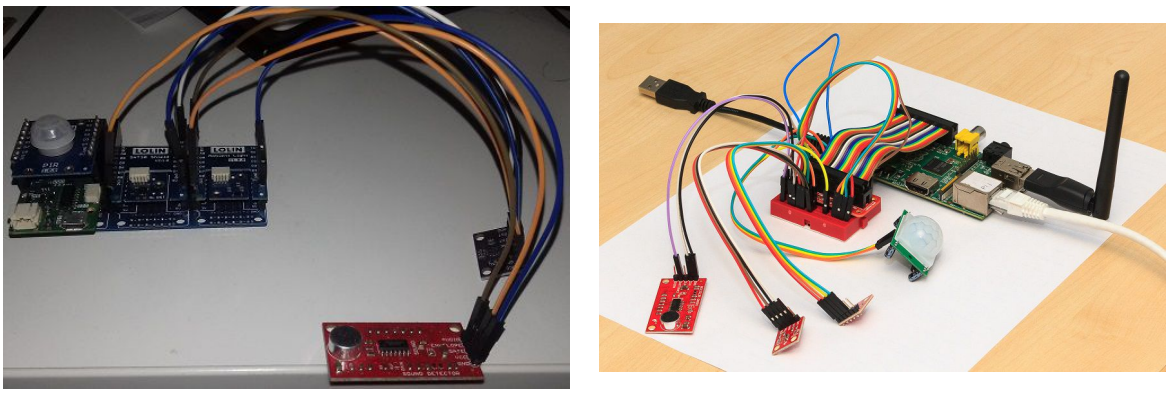

Figure 4.1: Exposed views of sensor nodes: the Wemos on the left is used in PWS and CWS; the Raspberry Pi on the right is used in PRS and CRS.

4.3 Smart Campus Case Study: Tiered Implementations

4.3.1 Sensor Node Hardware

The University of Glasgow has almost completed a ten year campus upgrade programme. A key goal is to have smart sensing infrastructure embedded into its new fabric. Prototype low-power and low cost commodity sensors for monitoring indoor environments are being utilised and evaluated. Currently, there is a small deployment of PRS implementations in 12 rooms for evaluation shown in Figure 4.1. However, to make the evaluation exercise more representative, the common ESP8266X Wemos D1 Mini microcontroller is selected for implementation of PWS shown in Figure 4.1.

Commodity sensors of a similar range are connected to both sensor nodes. The sensors are connected using a range of standard interfaces e.g. general purpose input/output pins (GPIO), IIC, SPI and one-wire. The sensors are as follows: Temperature & Humidity: LOLIN SHT30; Light: LOLIN BH1750; Motion: LOLIN PIR; Sound: SparkFun SEN-12642; CO2: SparkFun CCS811 [88].

4.3.2 PRS Implementation

Tiered IoT applications are often developed using C/C++ (Section 2.4.3). However, other languages like Python can be utilised. In fact, Python is now considered one of the top programming languages of choice in the development of IoT applications [21].

PRS is the first resource rich UoG smart campus IoT prototype that has been engineered and is Python based. The sensor nodes are connected to a RaspberryPi. This is a mass produced, portable and low powered computing device that has sold more than eight million devices worldwide [59].

The motherboard in this device is composed of several interfaces like I2C and USB for connecting the sensor nodes shown in Figure 4.1. Appendix D.1 provides an example of the

code to allow the light sensor node to interface with the RaspberryPi.

Several of the sensors are attached together to create a supersensor. This allows for the capturing of several types of environmental data in real time on one device. The sensor data is then periodically flushed to a remote server via the TCP/IP protocol [59].

4.3.3 PWS Implementation

In contrast to PRS that uses resource-rich Raspberry Pis, PWS (Figure 4.1) is a resource constrained microcontroller running MicroPython, a dialect of Python specifically designed to run on small, low powered embedded devices [72]. MicroPython has been developed as a variant of Python specifically for microcontrollers and embedded devices.

For developers, creating and coding during an interactive session with the hardware is as simple as using a serial terminal connection or using the WebREPL. However the use of IDEs like uPyCraft and Thonny MicroPython are also available [154]. It is pragmatic that MicroPython be utilised for the resource-constrained PWS as it minimises changes to the resource-rich PRS code base [88]. Appendix D.2 provides an example of the light sensor implementation for PWS.

4.3.4 Comparing PRS & PWS Implementations

The tiered PRS and PWS IoT stack implementations have been carefully designed to be similar to facilitate comparison. They both share the same server code shown in the code snippet in Listing 4.1. Sensor data is collected and stored in two database systems i.e. Redis and MongoDB as shown in Figures 2.6 & 4.7. Both run on the same server. Redis is used as a caching and queuing system while MongoDB acts as a persistent data store.

Listing 4.1: Sensor Data Upload from Redis to Webpage

```
1
2  def main():
3      # Get configuration values
4      config.init('measurements')
5      port = (config.get('Measurements', 'Host'),
           config.getint('Measurements', 'Port'))
6
7      mongo_host = config.get('Mongo', 'Host')
8      mongo_port = config.getint('Mongo', 'Port')
9      mongo_database = config.get('Mongo', 'Database')
10
11     redis_host = config.get('Redis', 'Host')
12     redis_port = config.getint('Redis', 'Port')
```

```
13
14      # Start Mongo Sender
15      mongo_sender = MongoMeasurementsSender(mongo_host,
           mongo_port, mongo_database)
16      mongo_sender.Start()
17
18      # Start Redis Sender
19      redis_sender = RedisMeasurementsSender(redis_host,
           redis_port)
20      redis_sender.Start()
21
22      # Initialize servicer and register received callbacks for
           each of the senders
23      servicer = MeasurementsService()
24      servicer.AddReceivedCallback(mongo_sender.ReceivedCallback)
25      servicer.AddReceivedCallback(redis_sender.ReceivedCallback)
26
27      # Start the TCP server
28      server = implementations.create_server(servicer)
29      server.add_insecure_port("%s:%d" % port)
30      server.serve_forever()
```

Data can be monitored via the webpage component in real time. It is provided through the websocket server that connects to the Redis database shown in the code snippet in Listing 4.2. Furthermore, current and historical sensor data can be polled through the HTTP REST API that connects to the MongoDB database shown in the code snippet in Listing 4.3.

Listing 4.2: Sensor Data Upload from Redis to Webpage

```
1
2  for message in p.listen():
3
4      if message['type'] not in ['message', 'pmessage']:
5          continue
6
7      try:
8          status =
              collector_pb2.SensorStatus.FromString(message['data'])
9          hostname = status.hostname
10         sensor_type =
              sensor_types.sensor_type_name(status.sensor_type)
```

```
11
12          key = "%s/%s" % (hostname, sensor_type)
13
14          sensor_value = str(round(status.float_value, 2) or
               status.int_value or status.bool_value)
15          data[key] = "%s,%s" % (status.timestamp, sensor_value)
16          except Exception as e:
17
18          print e
19          continue
20
21          if (time.time() - last_upload > upload_interval):
22
23              signature = sign_data(data, secret_key)
24              url = '%s/%s' % (upload_url, signature)
```

Listing 4.3: Code Snippet showing MongoDB Database Connection

```
1
2  class MongoMeasurementsSender(MeasurementsSender):
3      db = None
4
5      def __init__(self, mongo_host, mongo_port, mongo_database):
6
7          MeasurementsSender.__init__(self)
8          self.db = pymongo.MongoClient(mongo_host,
               mongo_port)[mongo_database]
9
10     def Send(self, hostname, s):
11
12         try:
13             t = collector_pb2.SensorStatus
14             #if it is an occupancy reading insert in the
                   occupancy collection
15             if s.sensor_type == t.OCCUPANCY:
16                 self.db.RoomOccupancy.insert({
17                     'timestamp': s.timestamp,
18                     'node_name': hostname,
19                     'value': s.bool_value,
20                     'predictor_hash': s.hash
```

```
21                 })
22             else:
23
24                 self.db.SensorReadingCollection.insert({
25                     'timestamp': s.timestamp,
26                     'value': s.int_value or s.float_value or
                           s.bool_value,
27                     'node_name': hostname,
28                     'sensor_type':
                           sensor_types.sensor_type_name(s.sensor_type)
29                     })
30         except Exception as e:
31             print "Failed to send to Mongo, discarding status.
                   %s" % e.message
```

Both tiered implementations also share the same object oriented paradigm with similar classes. For example, in PRS, Listing 4.4 shows a class for accessing the memory area and returning temperature sensor data. If any problems are encountered, a message is printed out. PWS has a similar class shown in Listing 4.5. Appendix D.3 provides the full code base for both implementations of the temperature sensor.

Listing 4.4: Code Snippet for PRS Temperature Sensor

```
1
2 class TemperatureSensor():
3
4 controller = implementations.create_controller()
5         results = TempConnect.temp_results(self)
6
7         if results != None and len(results.data_read) ==
              len(transaction.commands):
8             data = self._swap_word_bytes(results.data_read[0])
9             return (data >> 4) * 0.0625
10        else:
11            print "Problem with response from i2c service: %s"
                  % results
12            return None
```

Listing 4.5: Code Snippet for PWS Temperature Sensor

```
1
2 def get_sensor_value(self):
```

```
3
4           if results != None:
5
6               buf = self.mem1
7               buf[0] = self.reg1
8               buf[1] = self.reg2
9               results.writeto(addr, buf)
10              time.sleep_ms(1000)
11
12              data = self.mem2
13              buf[0] = self.reg3
14              buf[1] = self.reg4
15              results.writeto(addr, buf)
16              data = results.readfrom(addr, 6)
17
18              data2 = self._swap_word_bytes(data)
19              return data2
20
21          else:
22
23               print ("Problem with response from i2c service:
                    %s" % results)
24               return None
```

4.4 Understanding Tierless Clean iTasks/mTasks in IoT Programming

The following section is based on work conducted by Mart Lubbers and Pieter Koopman at Radboud University Nijmegen in the Netherlands. It forms part of the collaborative work coauthored and published in [87, 88].

4.4.1 Task Oriented Programming (TOP)

Task Oriented Programming (TOP) is a technique for constructing distributed systems through a declarative programming paradigm. It utilises the concept of *tasks*. This represents the building blocks of everything that needs to be done in order for the system to achieve its overall business function. For example, a task could involve a user having to complete a form, a database having to be monitored or another task having to be moderated [110].

The language implementation from TOP automatically creates an *integrated distributed system*. A single declarative description of tasks can generate all the necessary components like web servers and client code for browsers. This is accomplished through three key concepts: tasks, task combinators & Shared Data Stores (SDSs).

Tasks are basic functionalities that are needed for the system to achieve its overall business function. Examples include tasks for controlling peripherals like a servo motor or reading some IoT sensor data. Task combinators are for more advanced tasks. They allow task values to be observed by other tasks and for them to either be executed sequentially or in parallel. Examples include tasks being returned as a result of a function or tasks being repeated due to recursion [87, 88].

Shared Data Stores (SDSs) allow tasks to exchange information. This is accomplished through tasks being able to atomically observe and change the value of the typed SDS. Basically, there is an abstraction of data shared by different tasks similar to variables and persistent values.

4.4.2 Task Oriented Programming in Clean

Similar to Haskell, Clean [1]is statically typed, pure and non strict [1]. It enables functions to work on many types. However, in contrast to Haskell, Clean has a unique type system that allows the single threaded use of stateful objects like files and windows [10, 125]. Furthermore, it supports many models of generic programming. This is heavily utilised in TOP especially in the creation of web editors and communication protocols that work for any user-defined datatype.

Clean iTask/mTask are two DSLs embedded in Clean. They ensure bidirectional data sharing and automatic communication between the sensor nodes and server. A single declarative paradigm is utilised across all tiers in an IoT stack.

4.4.3 The iTasks Embedded DSL

Clean iTasks can be deployed on resource-rich sensor nodes. For example, Listing 4.6 provides a complete program where the room temperature can be repeatedly read from a digital and humidity temperature sensor (DHT). The values are displayed on a webpage shown in Figure 4.2. However, it only reports instantaneous temperature measurements. There is no state or recording of data.

This program can be extended to record and store temperatures. The times and the measurements that are taken can also be recorded. This creates a small tierless application displayed in the code in Listing 4.7.

There is a persistent SDS to store the temperature values recorded and for automatic communication between tasks. The *measurementsSDS:measureTask* tasks facilitate this everytime a

[1]Source: `https://clean-lang.org/`

Listing 4.6: SimpleTempSensor: a CleaniTask program to read a local room temperature sensor and display it on a web page

```
1  module simpleTempSensor
2
3  import iTasks
4
5  Start :: *World → *World (#λlabel{lst:iTaskTemp:systemfro}#)
6  Start world = doTasks readTempTask world
       (#λlabel{lst:iTaskTemp:systemto}#)
7
8  readTempTask :: Task Real
9  readTempTask =
10     withDHT IIO_TempID λdht →
           (#λlabel{lst:iTaskTemp:dhtDef}#)
11     repeatEvery interval (
           (#λlabel{lst:iTaskTemp:repeat}#)
12          temperature dht >>∼ λtemp →
                (#λlabel{lst:iTaskTemp:readDHT}#)
13          viewInformation [] temp <<@
                (#λlabel{lst:iTaskTemp:viewInformation}#)
14          Label "Temperature"
                (#λlabel{lst:iTaskTemp:label}#)
15     )
```

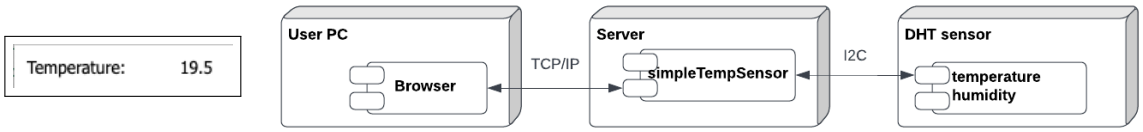

Figure 4.2: iTask SimpleTempSensor: Webpage (left) and Deployment Diagram (right).

Listing 4.7: TempHistory: A Tierless Clean iTask web application that records and manipulates timed temperatures

```
1  module TempHistory
2
3  import iTasks, iTasks.Extensions.DateTime
4
5  :: Measurement = {time :: Time, temp :: Real}
      (#λlabel{lst:iTaskTemp:Measurement}#)
6  derive class iTask Measurement (#λlabel{lst:iTaskTemp:derive}#)
7
8  measurementsSDS :: SimpleSDSLens [Measurement]
      (#λlabel{lst:iTaskTemp:measurementsSDS}#)
9  measurementsSDS = sharedStore "measurements" []
10
11 measureTask :: Task () (#λlabel{lst:iTaskTemp:measureTask}#)
12 measureTask =
13     withDHT IIO_TempID λdht →
14  let task old =
15    temperature dht >>⋆ (#λlabel{lst:iTaskTemp:step}#)
16    [ OnValue (ifValue ((≠) old) λtemp →
         (#λlabel{lst:iTaskTemp:action1}#)
17     get currentTime >>∼ λtime →
18     upd (λlist.[{time=time, temp=temp}:list]) measurementsSDS
          @! temp)
19    , OnValue (always (waitForTimer False interval @! old))
         (#λlabel{lst:iTaskTemp:action2}#)
20    ] >>∼ task
21  in task initialTemp (#λlabel{lst:iTaskTemp:launch}#)
22
23 controlSDS :: Bool → Task [Measurement]
24 controlSDS byTemp =
25  ((Label "# to take"    @>> enterInformation []) -||
      (#λlabel{lst:iTaskTemp:enter}#)
26   (Label "Measurements" @>> (#λlabel{lst:iTaskTemp:view}#)
27       viewSharedInformation [ViewAs (if byTemp (sortBy (λx
            y→x.temp < y.temp)) id)]controlSDS byTemp))
28   ,OnAction (Action (if byTemp "Sort time" "Sort temp"))
        (always (controlSDS (not byTemp)))
29   ] (#λlabel{lst:iTaskTemp:actionend}#)
30
31 mainTask :: Task [Measurement]
32 mainTask = controlSDS False -|| measureTask
```

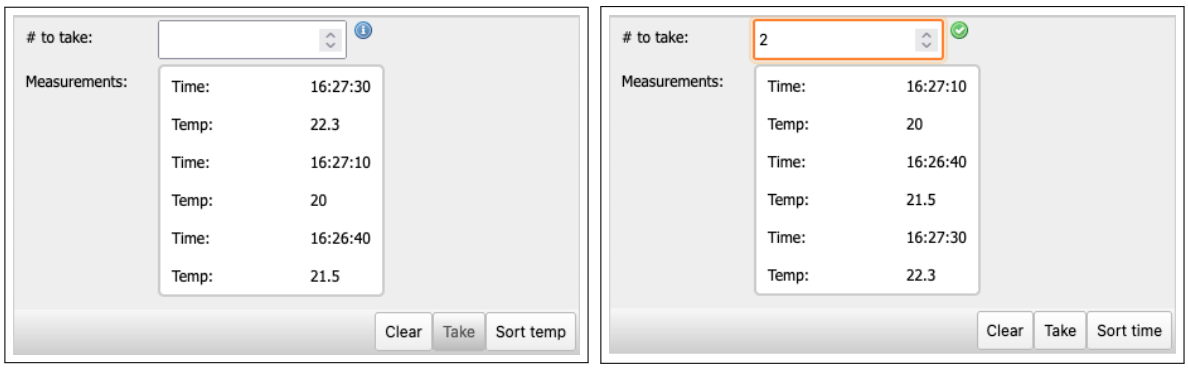

Figure 4.3: Web pages generated by the `TempHistory` tierless web application: on the left sorted by time; on the right sorted by temperature. The `Take` button is only enabled when the topmost editor contains a positive number.

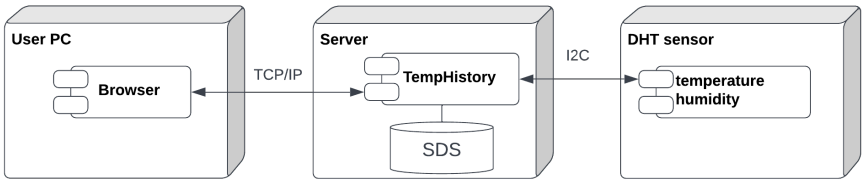

Figure 4.4: Deployment diagram of the iTask TempHistory tierless web application from Listing 4.7.

temperature change is detected. The *controlSDS* task allows a webpage to be generated and for users to control the view of the temperature measurements shown in Figures 4.3 & 4.4.

4.4.4 Engineering Tierless IoT Systems with iTask

Clean iTasks can also be used to support an iTask server on a resource rich device like a RaspberryPi. Listing 4.8 provides an example showing the temperature sensing system with a server and a single sensor node (CRTS (Clean RaspberryPi Temperature Sensor)).

This application is similar to Listing 4.7. However, the main differences are as follows. (1) The *devTask* task executes on the sensor node. (2) The recorded temperatures are timestamped on the SDS through the *dateTimeStampedShare latestTemp* task. (3) Temperature measurements are stored as tuples instead of tailor-made records. Figure 4.5 provides an example of how CRTS functions and how the temperature values are displayed on the webpage.

4.4.5 The mTask embedded DSL

Many IoT systems consist of sensor nodes that utilise microcontrollers. These devices are resource-constrained but provide significant advantages as follows when compared to the resource-rich RaspberryPi [88]. Microcontrollers are (1) far cheaper (2) consume less power (3) allow the programmer to easily control sensors and actuators through the IO pins of the processor.

Listing 4.8: CRTS: a tierless temperature sensing IoT system. Written in Clean iTask, it targets a resource-rich sensor node.

```
1  tempSDS :: SimpleSDSLens [(DateTime, Real)]
2  tempSDS = sharedStore "temperatures" []
3
4  latestTemp :: SDSLens () (? (DateTime, Real)) (DateTime, Real)
5  latestTemp = mapReadWrite (listToMaybe, λx xs→ ?Just [x:xs])
       ?None tempSDS
6
7  devTask :: Task DateTime
8  devTask = (#λlabel{lst:itaskTempFull:sensorfro}#)
9   withDHT IIO_TempID λdht →
10  forever ((#λlabel{lst:itaskTempFull:forever}#)
11   temperature dht >>∼ λtemp →
12   set temp (dateTimeStampedShare latestTemp) >-|
13   waitForTimer False interval)
         (#λlabel{lst:itaskTempFull:waitForTimer}#)
14   (#λlabel{lst:itaskTempFull:sensorto}#)
15
16 mainTask :: Task ()
17 mainTask (#λlabel{lst:itaskTempFull:main}#)
18  =    asyncTask deviceInfo.domain deviceInfo.port devTask
        (#λlabel{lst:itaskTempFull:startdevtask}#)
19   -|| viewSharedInformation []
        (#λlabel{lst:itaskTempFull:displaystart}#)
20   (remoteShare latestTemp deviceInfo)
         (#λlabel{lst:itaskTempFull:remoteShare}#)
21   <<@ Title "Latest temperature"
         (#λlabel{lst:itaskTempFull:displayend}#)
```

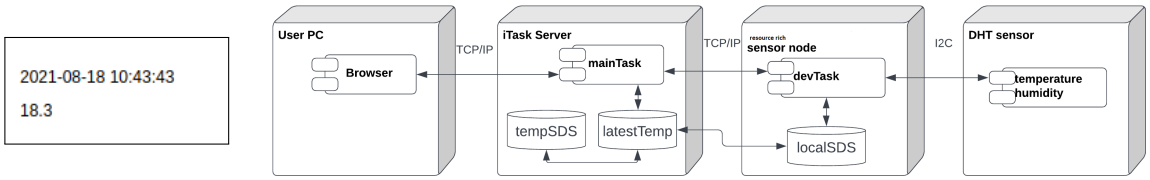

Figure 4.5: Tierless iTask CRTS temperature sensing IoT system: web page (left) and deployment diagram (right).

Executing Clean iTask on microcontrollers is difficult. Microcontrollers have limited memory capacity, compute power and communication bandwidth. Furthermore, there is no standard OS like Windows or Linux to run iTasks. Even if this is possible, execution of the code will be too slow given the limited processing capacity [88].

Clean mTask overcomes these challenges. It consists of a lightweight domain specific OS module that runs on microcontrollers. This module provides the following functionalities. (1) It facilitates the execution of mTask programs by receiving bytecode generated from those programs. (2) It manages updates to the SDS and the passing of results.

The OS is stored in flash memory and executes tasks in RAM. This slows the degradation of the microcontroller. For example, programs implemented using C/C++ for microcontrollers are stored and executed in flash memory. Constant uploading will eventually decrease the lifetime of the microcontroller as the flash memory will eventually become exhausted [88].

4.4.6 Engineering Tierless IoT Systems with mTasks

Clean mTask programs are normally implemented in the Perception and Network Layers of an IoT stack. Here, the programmer can select or customise tasks to be dynamically compiled at runtime. Furthermore, Clean iTasks can be integrated with mTasks as iTasks can be used to implement the Presentation & Application Layers of the stack. This is shown with the Clean Wemos Temperaturew Sensor (CWTS) implementation in Listing 4.9.

The *mainTask* is a simple iTask task that starts the *devTask* mTask task on the device. It reads temperature values from the *latestTemp* SDS that is shared with mTask device. The values are then displayed on a webpage shown in Figure 4.6.

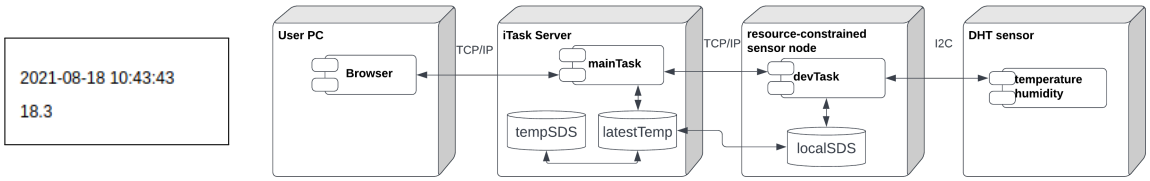

Figure 4.6: Tierless iTask/mTask CWTS temperature sensing IoT system: web page (left) and deployment diagram (right).

Listing 4.9: CWTS: a tierless temperature sensing IoT system. Written in Clean iTask mTask, it targets a resource-constrained sensor node.

```
1  module cwts
2
3  import mTask.Language, mTask.Interpret,
     mTask.Interpret.Device.TCP
4
5  import iTasks, iTasks.Extensions.DateTime
6
7  deviceInfo = {TCPSettings|host="...", port=8123, pingTimeout=
     ?None}
     (#λlabel{lst:mtasktemp:co0}#)  // CO
8  interval = lit 10

     (#λlabel{lst:mtasktemp:sn0}#)   // SN
9  DHT_pin = DigitalPin D4

     (#λlabel{lst:mtasktemp:si0}#)  // SI
10
11 Start world = doTasks mainTask world

     (#λlabel{lst:mtasktemp:wi0}#)  // WI
12
13 tempSDS :: SimpleSDSLens [(DateTime, Real)]
14 tempSDS = sharedStore "temperatures" []

     (#λlabel{lst:mtasktemp:di0}#)   // DI
15
16 latestTemp :: SDSLens () (? (DateTime, Real)) (DateTime, Real)
17 latestTemp = mapReadWrite (listToMaybe, λx xs→ ?Just [x:xs])
     ?None tempSDS
     (#λlabel{lst:mtasktemp:di1}#)// DI
18
19 devTask :: Main (MTask v Real) | mtask, dht, liftsds v

     (#λlabel{lst:mtasktemp:devTask}#)
20 devTask =
21   DHT (DHT_DHT DHT_pin DHT11) λdht →

       (#λlabel{lst:mtasktemp:DHT}#)(#λlabel{lst:mtasktemp:si1}#)
        // SI
22   liftsds λlocalSds =
23
24 mainTask :: Task Real
25 mainTask
26   =   withDevice deviceInfo λdev → liftmTask devTask dev
27   -|| viewSharedInformation [] latestTemp
```

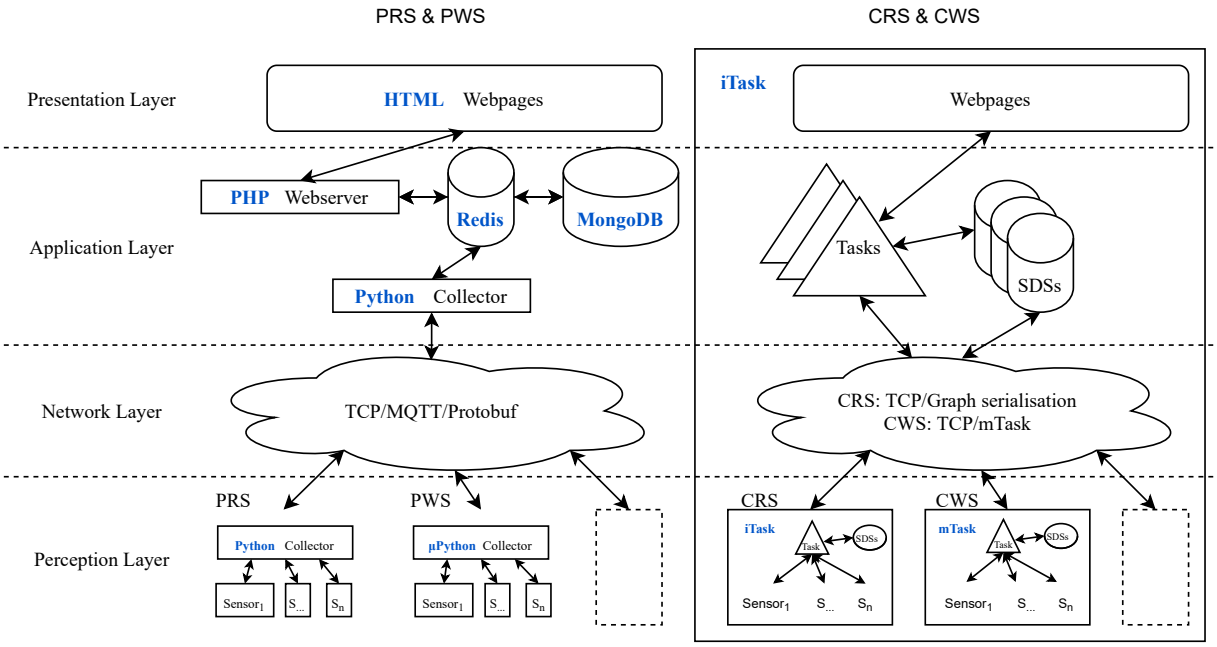

Figure 4.7: PRS and PWS (left) together with CRS and CWS (right) mapped to the 4-tier IoT architecture. Every box is the diagram denotes a source file or base. Bold blue text describes the language or technology used in that source. The Network and Perception Layers are unique to the specific implementation, where the Application and Presentation Layers are shared between implementations.

4.5 Smart Campus Case Study: Tierless Implementations

This section demonstrates the operational equivalence of the PRS, PWS, CRS and CWS implementations. CRS & CWS were implemented by Mart Lubbers and Pieter Koopman. We measured the memory residencies of the four applications. (Section 4.5.4).

4.5.1 Tierless Implementations: CRS & CWS

The second part of the UoG Smart Campus project involves comparing the tiered PRS and PWS against tierless implementations: CRS & CWS shown in Figure 4.7. Both tierless implementations are similar in some aspects.

They both share the same iTask server code and use a SQLite backend. Communication between a sensor node and server is initiated by the server. However, CRS executes Clean iTasks code as the sensor node is the RaspberryPi. In contrast, CWS integrates both iTask & mTask as the sensor nodes comprise the Wemos microcontroller [88].

4.5.2 Operational Equivalence

A fair comparison of all four IoT stack implementations require that they meet the functional requirements specified by the UoG project board. The system should:

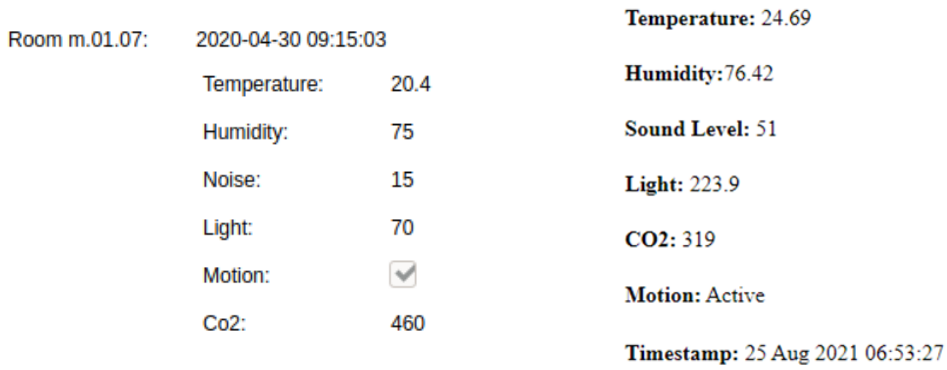

Figure 4.8: Web interfaces for CWS and CRS (left); PWS and PRS (right).

(1) Be able to measure temperature and humidity as well as light intensity,

(2) Scale to no more than ten sensors per sensor node and investigate further sensor options like measuring sound levels,

(3) Have access to communication channels like WiFi, Bluetooth and even wired networks.

(4) Have a centralised database server,

(5) Have a client interface to access information stored in the database,

(6) Provide some means of security and authentication,

(7) Have some means of managing and monitoring sensor nodes like updating software or detecting new sensor nodes.

All four smart campus implementations meet these high-level requirements as illustrated in Figure 4.7.

4.5.3 Functional Equivalence

All four implementations utilise the same inexpensive sensors. It is expected that the sensor data collected like temperature, humidity, motion, light and sound will be within tolerance levels. Figure 4.8 shows the web interface that displays the sensor readings for all implementations.

This is validated by comparing PRS and PWS sensor nodes deployed in the same room for some minutes. The measurements show only small variances, e.g. temperatures recorded differ by less than $0.4°C$, and light by less than one lux. For this room monitoring application precise timings are not critical, and we do not compare the timing behaviours of the implementations [88].

4.5.4 Memory Consumption

Computational capacity is a key issue for sensor nodes. Limited memory affects processing capabilities. Supersensors have up to 1 GiB of memory while microcontrollers have just tens of KiBs. Determining memory efficiency for this evaluation is key because the tierless languages generate code for the sensor nodes to be executed.

Table 4.1 shows the maximum memory residency after garbage collection of the sensor node for all four smart campus implementations. The smart campus sensor node programs executing on the Wemos microcontrollers have low maximum residencies: 20270 bytes for PWS and 880 bytes for CWS. In CWS, the mTask system generates very high level TOP bytecode that is interpreted by the mTask virtual machine and uses a small and predictable amount of heap memory [88].

In PWS, the hand-written MicroPython is compiled into bytecode for execution on the virtual machine. Low residency is achieved with a fixed size heap and efficient memory management. For example both MicroPython and mTask use fixed size allocation units and mark & sweep garbage collection to minimise memory usage at the cost of some execution time [117].

Table 4.1: UoG Smart Campus Sensor Nodes: Maximum Memory Residency (Bytes).

PWS	PRS	CWS	CRS
20,270	3,557,806	880	2,726,680

The smart campus sensor node programs executing on the Raspberry Pis have far higher maximum residencies than those executing on the microcontrollers: 3.5MiB for PRS and 2.7MiB for CRS. In CRS the sensor node code is a set of iTask executing on a full-fledged iTask server running in distributed child mode and this consumes far more memory.

In PRS the sensor node program is written in Python, a language far less focused on minimising memory usage than MicroPython. For example an object like a string is larger in Python than in MicroPython and consequently does not support all features such as *f-strings* [88]. Furthermore, not all advanced Python features regarding classes are available in MicroPython, i.e. only a subset of the Python specification is supported [162].

In summary, iTask and mTask code are memory efficient for the sensor nodes. Generally, the maximum residencies for the Clean code is even less than the hand written tiered implementations. However, hand-written code can be made more memory efficient. One approach is to use a more memory efficient language like C/C++. However, this may involve more developer effort and could introduce additional challenges like semantic friction [88].

4.5.5 Power Consumption

Sensor nodes are usually designed for low power consumption. This is crucial especially if IoT systems are utilising batteries. The Wemos sensor nodes used in CWS and PWS have the

low power consumption of a typical embedded device: with all sensors enabled, they consume around 0.2W. The Raspberry Pi supersensor node used in CRS and PRS use more power as they have a general purpose ARM processor and run mainstream Linux. With all sensors enabled, they consume 1–2W, depending on ambient load. So a microcontroller sensor node consumes an order of magnitude less power than a supersensor node [88].

4.6 Evaluation

In this section, my PWS implementation together with Mart's CWS implementation provide the basis for the Section 4.6.2 comparison of IoT programming for resource-constrained sensor nodes with programming for resource-rich IoT nodes (as in PRS and CRS). In Section 4.6.1 while Mart Lubbers measured the CRS and CWS implementations, I measured the PRS and PWS implementations and contributed to the analysis of the result. In Section 4.6.6 I contributed to the analysis of the reliability of tiered and tierless IoT stacks, for example locating an instance where type safety is lost in PRS and PWS (Section 4.6.6).

4.6.1 Is Tierless IoT Programming Easier Than Tiered?

Code Size is a common metric that is used to measure developer and maintenance effort of a software system. It is based on the simple premise that the more source lines of code (SLOC) utilised in developing a program, the greater the developer effort and the more likelihood of bugs [129]. SLOC involves counting lines of code that are not comments or blank lines.

This metric offers benefits like being a simple measure, not being dependent on some formula and can automatically be computed [136]. However, some drawbacks are that SLOC can be influenced by programming style, language paradigm and counting methods [3].

Table 4.2 enumerates the SLOC required to implement the UoG smart campus functionalities in PWS, PRS, CWS and CRS. Both Python and Clean implementations use the same server and communication code for Raspberry Pi and for Wemos sensor nodes. The Sensor Interface (SI) refers to code that allows communication between the peripherals and the sensor node software. Sensor Node (SN) code refers to all code on the sensor node that has not been categorised like control flow. Manage Nodes (MN) refer to code that facilitates coordination among the sensor nodes e.g. adding a new sensor to the system. Web Interface (WI) refers to the web interface from the server, i.e. the Presentation Layer. Database Interface (DI) is code that ensures communication between the server and the database(s). Communication (CO) code executes on both sensor node and server in order to facilitate communication between both devices at the Network Layer [88].

The tierless implementations require less code than the tiered implementations. For example 166/562 SLOC for CWS/PWS, or 70% fewer source lines. This can be attributed to reduced interoperation, automatic communication, and high level programming abstractions [88].

Table 4.2: Comparing tiered and tierless smart campus code sizes: SLOC and number of source files. PWS and CWS execute on resource-constrained sensor nodes, while PRS and CRS execute on resource-rich sensor nodes.

		Tiered Python		Tierless Clean	
Code Location	Functionality	PWS	PRS	CWS	CRS
Sensor Node	Sensor Interface	52	57	11	11
	Sensor Node	178	183	9	4
Server	Manage Nodes	76		35	30
	Web Interface	56		28	
	Database Interface	106		78	
Communication	Communication	94	98	5	4
Total SLOC		562	576	166	155
No. Files		35	38	3	3

Code Proportions Some of the issues of SLOC can be avoided by comparing the percentage of code required to implement the four IoT stacks. Figure 4.9 shows the percentage of the total SLOC required to implement the smart campus functionalities in each of the four implementations, and is computed from the data in Table 4.2.

It shows that there are significant differences between the percentage of code for each functionality between the tiered and tierless implementations. For example 17% of the tiered implementations specifies communication, whereas this requires only 3% of the tierless implementations, i.e. six× less. The reasons for this are explored in Section 4.6.4.

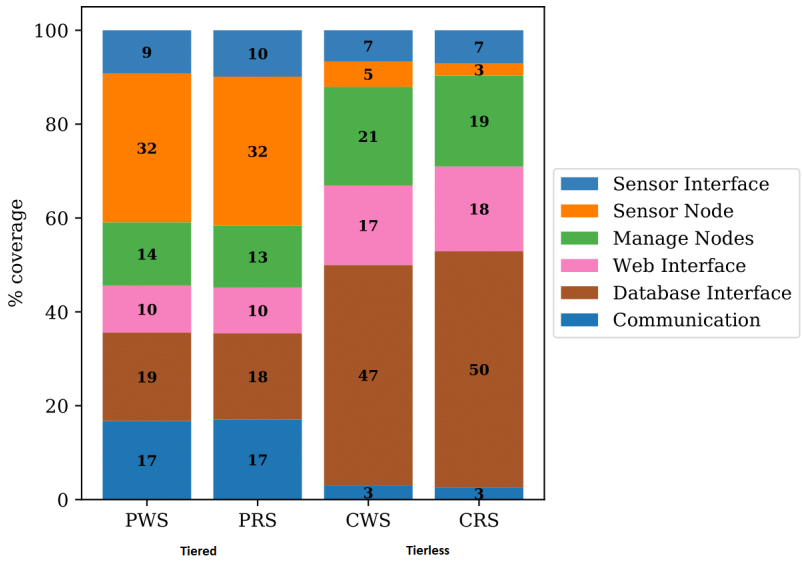

Figure 4.9: Comparing the percentage of code required to implement each functionality in tiered/tierless and resource-rich/constrained smart campus implementations.

The other major difference is the massive percentage of Database Interface code in the tier-

less implementations: at least 47%. A standard DBMS is required based on the Smart Campus specification requirements and the Clean iTask SQL interface requires only 78 SLOC. This is smaller than the 106 SLOC used in Python (Table 4.2). However, when converted to percentages, the tiered implementations value is far higher especially for systems with total codebases of only around 160 SLOC.

The reason for this is that idiomatic Clean/iTask would use high level abstractions to store persistent data in an SDS, requiring just a few SLOC. The total size of CWS and CRS could even be reduced by a factor of two and the percentage of Database Interface code would be even less than in the tiered Python implementations if more high level abstractions are applied [88].

4.6.2 Comparing Codebases for Resource-Rich/Constrained Sensor Nodes

Before exploring the reasons for the smaller tierless codebase, the implementations for resource-rich and resource-constrained sensor nodes are compared, again using SLOC and code proportions. Table 4.2 shows that the two tiered implementations are very similar in size: with PWS for microcontrollers requiring 562 SLOC and PRS for supersensors requiring 576 SLOC. The two tierless implementations are also similar in size: CWS requiring 166 and CRS 155 SLOC.

There are two main reasons for this similarity. One is that the server-side code, especially in the the Presentation and Application Layers, is identical for both resource rich/constrained implementations. This accounts for approximately 40% of the PWS and PRS codebases, and approximately 85% of the CWS and CRS codebases (Figure 4.9). For the Perception and Network Layers on the sensor nodes, the Python and MicroPython implementations utilise the same structure, e.g. a class for each type of sensor and analogous libraries. Indeed, approaches like CircuitPython [152] allow the same code to execute on both resource-rich and resource-constrained sensor nodes.

Similarly, like the Python based tiered implementations, iTask and mTask are designed to be similar, as elaborated in Section 4.4. The similarity is especially observed when comparing the iTask CRTS and iTask/mTask CWTS room temperature systems in Listings 4.8 and 4.9. Both implementations use similar SDS data stores and lenses. They have similar `devTasks` that execute on the sensor node, and the server-side `mainTasks` are almost identical: they deploy the remote `devTask` before generating the web page to report the readings.

In both Python and Clean the resource-constrained implementations are less than 7% larger than the resource-rich implementations. This suggests that *the development and maintenance effort of simple IoT systems for resource-constrained and for resource-rich sensor nodes are similar in tierless technologies,* just as it is in tiered technologies. A caveat is that the smart campus system is relatively simple, and developing more complex Perception and Network code on bare metal may prove more challenging. It is possible that the lack of OS support, and the use of both restricted languages and libraries may have greater impact.

4.6.3 Reduced Interoperation

Table 4.3: Smart Campus implementation languages and paradigm comparison.

		Languages				Paradigms	
Code Location	Functionality	PWSS	PRSS	CWSS	CRSS	Python	Clean
Sensor Node	Sensor Int.	μPython	Python	mTask	iTask	imp.	decl.
	Sensor Node	μPython	Python	mTask	iTask	imp.	decl.
Server	Manage Nodes	Python, JSON		iTask		imp.	decl.
	Web Int.	HTML, PHP		iTask		both	decl.
	Database Int.	Python,JSON,Redis		iTask		both	decl.
Communication	Communication	μPython	Python	iTask,mTask	iTask	imp.	decl.
	Total	7	6	2	1	2	1

The majority of tiered IoT systems are implemented using a number of different programming languages and paradigms. The components within these systems must have proper implementation to ensure efficient interoperation. However, tierless IoT implementations are usually simpler and shorter than the tiered implementations because they use far fewer programming languages and paradigms. In the Smart Campus Case Study, language is the main factor for distinguishing embedded DSLs from their host language. As a result, iTask and mTask are considered distinct from Clean; and to distinguish dialects: so MicroPython is considered distinct from Python [88].

The tierless implementations use just two conceptually similar DSLs embedded in the same host language, and a single paradigm (Table 4.3). In contrast, the tiers in PRS and PWS use six or more very different languages. Both consist of imperative and declarative paradigms.

The use of multiple languages in other typical software systems like web stacks is common. For example, a recent survey of open source projects reveals that on average at least five different languages are used [94]. However, interoperating components in multiple languages and paradigms raises a plethora of issues.

Interoperation *increases the cognitive load on the developer.* The person must simultaneously think in multiple languages and paradigms. This is one of the main contributing factors that can cause semantic friction or impedance mismatch [67]. A simple illustration of this is that the tiered PRS source code comprises some 38 source and configuration files, whereas the tierless CRS requires just three files (Table 4.2). The source could be structured as a single file. However, to enforce separate concerns, there are three modules, one each for SDSs, types, and control logic [145].

The developer must *correctly interoperate the components*, e.g. adhere to the API or communication protocols between components. The interoperation often entails additional programming tasks to specify the marshalling or demarshalling data between components. For example, in the tiered PRS and PWS architectures, JSON is used to serialise and deserialise data strings from the Python collector component before storing the data in the Redis database (Listing 4.10).

Listing 4.10: JSON Data marshalling in PRS and PWS: Sensor Node Above, Server Below.

```
1
2 channel = 'sensor_status.%s.%s' % (hostname,
3  sensor_types.sensor_type_name(s.sensor_type))
4   self.r.publish(channel, s.SerializeToString())
5
6 (#\dotfill#)
7
8 for message in p.listen():
9   if message['type'] not in ['message', 'pmessage']:
10  continue
11
12 try:
13  status =
      collector_pb2.SensorStatus.FromString(message['data'])
```

To ensure correctness the developer *must maintain type safety* across a range of very different languages and diverse type systems. This is explored further in Section 4.6.6. The developer must also potentially handle *diverse failure modes*, not only of each component, but also of their interoperation. For example, if a value of an unexpected type is passed through an API. This issue is explored further in Section 4.6.6.

4.6.4 Automatic Communication

In conventional tiered IoT implementations the developer must write and maintain code to communicate between tiers. For example PRS and PWS create, send and read MQTT [77] messages between the Perception and Application Layers. Table 4.2 shows that communication between these layers require some 94 SLOC in PWS and 98 in PRS, accounting for 17% of the codebase (bottom bars in Figure 4.9). To demonstrate this, Listing 4.11 shows part of the code to communicate sensor readings from the PWS sensor node to the Redis store on the server.

In tiered IoT implementations, the developer has to write additional code for communication because it is an intricate process. For example, in such a distributed system, the sender and receiver must be correctly configured, correctly follow the communication protocol through all execution states, and deal with potential failures. For example line 3 of Listing 4.11: `redis host = config.get('Redis', 'Host')` will fail if either the host or IP are incorrect.

Listing 4.11: Tiered Communication Example: MQTT Transmission of Sensor Values in PWSS.

```
1 def main():
2  config.init('mqtt')
3  redis_host = config.get('Redis', 'Host')
```

```
4  redis_port = config.getint('Redis', 'Port')
5  r = redis.StrictRedis(host=redis_host, port=redis_port)
6  p = r.pubsub()
7  p.psubscribe("sensor_status.*")
8  for message in p.listen():
9   if message['type'] not in ['message', 'pmessage']:
10   print "Ignoring message %s" % message
11   (#\ldots#)
```

In contrast, the tierless CWS and CRS communication is both highly automated and automatically correct because matching sender and receiver code is generated by the compiler. Table 4.2 shows that communication is specified in just five SLOC in CWS and four in CRS, or just 3% of the codebase (bottom bars in Figure 4.9).

Listing 4.9 illustrates communication in a tierless IoT language. That is, the CWTS temperature sensor requires just three lines of communication code, and uses just three communication functions. The `withDevice` function integrates a sensor node with the server, allowing tasks to be sent to it. The `liftmTask` integrates an mTask in the iTask runtime by compiling it and sending it for interpretation to the sensor node. The `liftsds` integrates SDSs from iTask into mTask, allowing mTasks to interact with data from the iTask server. The exchange of data, user interface, and communication are all automatically generated.

4.6.5 High Level Abstractions

Another reason that the tierless Clean implementations are concise is because powerful higher order IoT programming abstractions are implemented. To illustrate and provide a better understanding, the simple temperature sensor from Listing 4.9 is used to compare the expressive power of Clean and Python-based IoT programming abstractions. There are implementations for all 4 configurations: PRTS (Python Raspberry Pi Temperature Sensor)[2], PWTS (Python Wemos Temperature Sensor)[2] , CRTS[3] and CWTS[3] but as the programming abstractions are broadly similar, only the PWTS and CWTS implementations are compared.

Although the temperature sensor applications are small compared to the smart campus applications, they share some typical IoT stack traits. The architecture consists of a server and a single sensor node (Figure 4.6). The sensor node measures and reports the temperature every 10 seconds to the server while the server displays the latest temperature via a web interface to the user.

[2]Lubbers, M.; Koopman, P.; Ramsingh, A.; Singer, J.; Trinder, P. (2021): Source code, line counts and memory stats for PRS, PWS, PRT and PWT. Zenodo. 10.5281/zenodo.5081386. NOTE: the dataset will be uploaded, the DOI is reserved.

[3]Lubbers, M.; Koopman, P.; Ramsingh, A.; Singer, J.; Trinder, P. (2021): Source code, line counts and memory stats for CRS, CWS, CRTS and CWTS. Zenodo. 10.5281/zenodo.5040754. NOTE: the dataset will be uploaded, the DOI is reserved.

Table 4.4 compares the SLOC required for the MicroPython and CleaniTaskmTask Wemos temperature sensors: PWTS and CWTS respectively. However, the code sizes for these applications are not considered in comparing the programming models as implementing such a small application as a conventional IoT stack requires a significant amount of configuration and other machinery that would be reused in a larger application. Hence, the ratio between total PWTS and CWTS code sizes (298:15) is far greater than for realistic applications like PWS and CWS (471:166).

Table 4.4: Comparing Clean and Python Programming Abstractions Using the PWTS and CWTS Temperature Sensors (SLOC and Total Number of Files.)

Location	Functionality	PWTS	CWTS
Sensor Node	Sensor Interface	14	3
	Sensor Node	67	4
Server	Web Interface	17	3
	Database Interface	106	2
Communication	Communication	94	3
Total SLOC		298	15
No. Files		27	1

The multiple tiers in PRS and PWS provide different levels of abstraction and separation of concerns. However, there are various ways that high-level abstractions make the CWS much shorter than PRS and PWS implementations.

Firstly, functional programming languages are generally more concise than most other programming languages because their powerful abstractions like higher-order and/or polymorphic functions require less code to describe a computation. Secondly, the TOP paradigm used in iTask and mTask reduces the code size further by making it easy to specify IoT functionality concisely. As examples, the step combinator `>>*.` allows the task value on the left-hand side to be observed until one of the steps is enabled; and the `viewSharedInformation` (line 31 of Listing 4.9) part of the UI will be automatically updated when the value of the SDS changes. Moreover, each SDS provides automatic updates to all coupled SDSs and associated tasks. Thirdly, the amount of explicit type information is minimised in comparison to other languages, as much is automatically inferred [63].

4.6.6 Could Tierless IoT Programming Be More Reliable than Tiered?

This section investigates whether tierless languages make IoT programming more reliable. It can be argued that having a much smaller and simpler code base is inherently more understandable, and more likely to be correct in order to improve reliability. Here specific language issues are explored, namely preserving type safety and failure management.

Type Safety Strong typing is important in the development life cycle of an application because it identifies errors early, and hence plays a crucial role in improving software quality. As a result, many modern languages provide strong typing, and encourage static typing to minimise runtime errors. However, many distributed system components written in languages that primarily use static typing, like Haskell and Scala, also utilise some dynamic typing, e.g. to ensure that the data arriving in a message has the anticipated type [38, 53].

In a typical tiered multi-language IoT system the developer must interoperate software in different languages with very different type systems, and even potentially executing on different hardware. The challenges of maintaining type safety have long been recognised as a major component for semantic friction in multi-language systems, e.g. [67].

Even if the different languages used for component interoperation are both strongly typed, they may attribute, often quite subtly, different types to a value. Type errors like this can lead to runtime errors, or the application silently reporting erroneous data. Errors such as these can be hard to troubleshoot. Automatic detection of such errors is sometimes possible, but requires the use of additional tools like Jinn [42, 75]. This tool is implemented in the Java JVM to check for any language transition bugs or constraints between Java and C and automatically diagnoses the problem if any issues are detected [75].

Listing 4.12: PRS loses Type Safety as a Sensor Node sends a `double`, and the Server stores a `string`.

```
message SensorData {
 enum SensorType { TEMPERATURE = 1; (#\ldots#) }
 SensorType sensor_type   = 1;
 uint64     timestamp     = 2;
 double     float_value   = 3;
}
(#\dotfill#)
channel = 'sensor_status.%s.%s' % (hostname,
 sensor_types.sensor_type_name(s.sensor_type))
  self.r.publish(channel, s.SerializeToString())
```

Analysis of the PRS codebase reveals an instance where it, fairly innocuously, loses type safety. The fragment in Listing 4.12 first shows a `double` sensor value being sent from the sensor node, and then shows the value being stored in Redis as a `string` on the server. As PWS consists of similar components, it also suffers from the same loss of type safety.

A tierless language makes guarantees type safety across an entire IoT stack. For example the Clean compiler guarantees static type safety as the entire CWS software stack is type checked, and generated, from a single source. Tierless web stack languages like Links [27] and Hop [135] provide the same guarantee for web stacks.

Listing 4.13: An mTask Failover Combinator.

```
1 failover :: [TCPSettings] (Main (MTask BCInterpret a)) → Task
     a
2 failover []     _     = throw "Exhausted device pool"
3 failover [d:ds] mtask = try ( withDevice d (liftmTask mtask) )
     except
4 where except MTEUnexpectedDisconnect = failover ds mtask
5       except _                       = throw e
```

Failure Management Some IoT applications, including smart campus and other building monitoring applications, require high sensor uptimes. Hence, if a sensor or sensor node fails the Application Layer must be notified, so that it can report the failure. In the UoG smart campus system a building manager is alerted to replace the failed device.

In many IoT architectures, including PRS and PWS, detecting the reason for failure is challenging because the Application Layer listens to the devices. When a device comes online, it registered with the application and starts sending data. When a device goes offline again, it could be because the power was out, the device was broken or the device just paused the connection.

If a sensor node fails in CWS, the iTask/mTask combinator interacting with a sensor node will throw an iTask exception. The exception is propagated and a handler can respond, e.g. rescheduling the task on a different device in the room, or requesting that a manager replaces the device. That is, iTask, uses standard succinct declarative exception handling.

In the UoG smart campus application, this can be done by creating a pool of sensor nodes for each room and when a sensor node fails, assign another one to the task. Listing 4.13 shows a failover combinator that executes an mTask on one of a pool of sensor nodes. If a sensor node unexpectedly disconnects, the next sensor node is tried until there are no sensor nodes left. If other errors occur they are propagated as usual.

Currently, PRS and PWS both use heartbeats to confirm that the sensor nodes are operational, and will report failures. However, at the cost of extending the codebase, failover to an alternate sensor node cannot be provided.

4.6.7 Maintainability

Far more engineering effort is required in maintaining a system, than on the initial development. Tiered and tierless IoT systems have very different maintainability properties.

The modularity of the tiered stack makes replacing tiers/components easy. For example in PWS or PRS the MongoDB NoSQL DBMS could readily be replaced by an alternative like CouchDB. However, this is more difficult for the tierless CWS and CRS. A tierless compiler must generate code for components and replacing them may not be so easy.

If there are iTask abstractions for the component then replacement is straightforward. For

example replacing SQLite with some other SQL DBMS simply involves having to recompile the application. However incorporating a component that does not yet have a task abstraction, like a NoSQL DBMS, is more involved. That is, a foreign function interface to the new component must be implemented, along with a suitable iTask abstraction for operations on the component.

However, many maintenance tasks in tierless applications are smaller in scale and occur within the components or tiers. Consider a simple change, for example if the temperature value recorded by a sensor changes from integer to real. All tiers of a tiered stack must be correctly and consistently refactored to reflect the change of temperature data type: so changes at the Perception, Network, Application and Presentation Layers. A PWS developer works in seven languages and two paradigms to effect the change (Table 4.2), and must edit many source files. Many programming errors are either detected at runtime when testing the stack, or worse not automatically detected and produce erroneous results.

In a tierless language the source code is much smaller and so it is easier to comprehend, i.e. to understand what refactoring is required. A CWS developer works in only two languages and a single paradigm to effect the change, and will edit no more than three source files (Table 4.2). Moreover, the compiler will statically detect many programming errors.

More substantial in-component maintenance raises similar issues as for tiered implementations. If the maintenance activity requires a new task combinator, this is readily constructed in iTask, but may require changing the DSL implementation in mTask, i.e. to change the compiler and the bytecode interpreter. That is, mTask is more *brittle* than iTask. In summary, while a tiered approach makes replacing components easy, refactoring within the components is far harder in a multi-tier multi-language IoT implementations than in a tierless IoT implementation.

4.6.8 Support

Community and tool support are essential for engineering reliable production software. PRS and PWS are both Python based, and Python/MicroPython are among the most popular programming languages [21]. Python is also a common choice for some tiers of IoT applications [151]. Hence, there are a wide range of development tools like IDEs and debuggers, a thriving community and a wealth of training material. There are even specialised IoT Boards like PyBoard & WiPy that are specifically programmed using Python variations like MicroPython.

In contrast, tierless languages are far less mature than the languages used in tiered stacks, and far less widely adopted. This means that for CWS and CRS there are fewer tools, a far smaller developer community, and less training material available.

CWS and CRS are both written in DSLs embedded in Clean, a fairly stable industrial-grade but niche functional programming language. The DSLs are implemented in Clean but require experimental compiler extensions that are often undocumented. There are few maintainers of the DSLs and documentation is often sparse. Acquiring information about the systems requires distilling academic papers and referring to the source code. There is a Clean IDE, but it does not

contain support for the iTask or mTask DSLs.

4.6.9 Evaluation Summary & Reflection

Summary The key findings from this evaluation are as follows. (1) Tierless developers do indeed manage less interoperation: CRS uses a single DSL and paradigm, and CWS uses two DSLs in a single paradigm and three source code files. In contrast, both PRS and PWS use at least six languages in two paradigms and spread over at least 35 source code files (Tables 4.2 and 4.3). (2) A tierless stack provides improved reliability as it minimises semantic friction. (3) Tierless developers benefit from automatically generated, and hence correct, communication (Listing 4.9), and write a sixth of the amount of communication code (Figure 4.9). (4) The tierless implementations have very similar code sizes (SLOC), as do the tiered implementations: less than 7% difference in Table 4.2. This suggests that the development and maintenance efforts of simple tierless IoT systems for resource-constrained and for resource-rich sensor nodes are similar, as it is for tiered technologies.

Reflection This study has investigated some of the potential benefits of tierless languages for IoT systems. An IoT system implemented with a single tierless program is amenable to a host of programming language technologies. For example, if the language has a formal semantics, as Links, Hop and Clean tasks do, it is possible to prove properties of the system, e.g. [27,117,135].

However, tierless languages do have some challenges. There is a learning curve as programmers must master new tierless programming abstractions. Furthermore, the semantics of these automatic multi-tier behaviours are necessarily relatively complex. For example, in Clean, this entails becoming proficient with the iTask and mTask DSLs.

Moreover, specifying a behaviour that is not already provided by the tierless language requires either a workaround, or extending a DSL. However, implementing the relatively simple smart campus application required no such adaption. Above all else, tierless IoT technology is very new. The tools and community support have yet to mature when compared to mainstream programming languages used in tiered software stack implementation.

4.7 Chapter Summary

This chapter reports *the first comparison of interoperation in tiered and tierless IoT stacks through the systematic evaluation of four implementations of the prototype UoG smart campus IoT system: two tierless and two Python-based tiered.* The two tiered Python implementations are (PRS & PWS). The tierless implementations are Clean tierless (CRS & CWS). They are all operationally equivalent and meet the the UoG smart campus functional requirements (Section 4.5.2).

It is shown that *tierless languages have the potential to significantly reduce the development effort for IoT systems* (Section 4.6.1). Analysis of the code reduction reveals the following. (1) Tierless developers manage less interoperation. (2) Tierless developers benefit from automatically generated, and hence correct, communication. (3) Tierless developers can exploit powerful high-level declarative and task-oriented IoT programming abstractions.

Furthermore, it is also shown that *tierless languages have the potential to significantly improve the reliability of IoT systems* (Section 4.6.6). However, maintainability efforts in a tiered and tierless approach are similar (Section 4.6.7) while there is more community support for mainstream tier based programming languages (Section 4.6.8).

While offering real benefits for IoT systems development, tierless languages also raise some challenges. Programmers must master new tierless programming abstractions, and the semantics of these automatic multi-tier behaviours are necessarily relatively complex. In the Clean context this entails becoming proficient with the iTask and mTask DSLs. Moreover, specifying a behaviour that is not already provided by the tierless language requires either a workaround, or extending a DSL. However, implementing the relatively simple smart campus application required no such adaption. Finally, tierless IoT technology is very new, and both tool and community support have yet to mature.

Chapter 5

Could Eliminating Tiers Improve Interoperation?

Web applications are structured as distributed multi-tier stacks of components. Each component may be written in a different language, hosted in its own node and interoperate using a variety of protocols. Such interoperation increases developer effort, may reduce performance and requires additional resources. A range of approaches have been explored to improve interoperation but challenges still remain.

This chapter explores a pragmatic approach to improve interoperation in distributed multi-tier software stacks. The technique involves eliminating a component. Specifically, an evaluation is conducted to explore the implications of eliminating the Apache web server tier in a 4-tier Jupyter Notebook web stack (JAPyL) and replacing it with PHP libraries within the webpage component. The study systematically investigates the implications for web stack performance, resource consumption and programming effort.

Section 5.1 justifies why the tier elimination approach is necessary. Section 5.2 describes the theory of how tier elimination works. Section 5.3 discusses how 3-tier JPL was implemented. Section 5.4 compares interoperation in 3-tier JPL to 4-tier JAPyL. Section 5.5 summarises the findings from the evaluations and what generalisations can be made. Finally, Section 5.6 summarises the entire chapter.

5.1 Motivation

A range of approaches have been used to improve interoperation in web stacks. Some stacks focus on a single language, e.g. MEAN focuses on Javascript (Section 2.4.1). Other stacks use a common VM to minimise the interoperation overheads between languages, e.g. the .NET framework uses the Common Language Runtime (CLR) (Section 2.4.1). Sometimes web stack languages are combined, e.g. PyHyp combines Python and PHP [11]. The most radical approach is to combine all web stack languages into a single *tierless* language like Links or Hop

(Section 2.4.3).

However, these techniques do not entirely eliminate all factors responsible for interoperability challenges in multi-tier software stacks. For example, alternate solutions like the MEAN web stack and chained microservices are still prone to tight coupling that causes catastrophic failure discussed in Sections 2.4.1, 3.9.4 & 2.3.2.

Tierless languages like iTasks and mTasks in Sections 4.4 do indeed improve interoperation. They reduce the polyglot of languages when compared to tiered implementations like PRS & PWS. However, they are a new technology. Developers must effectively still use sophisticated language technologies like Clean iTask and mTask DSLs in order to implement these software stacks. In addition, there are a few instances where tierless languages require more developer effort when compared to mainstream programming languages discussed in Section 2.4.3 & Appendix B.2.

An alternative approach for improving interoperation is to eliminate *one* of the tiers in a distributed multi-tier software stack. Here, a component is eliminated and its functionality integrated into another existing component.

5.2 The Theory of Tier Elimination

5.2.1 Component Modularity

Modularity facilitates a top down approach in organising components in a distributed multi-tier software stack [61]. For example, in JAPyL (Figure 1.4), neither the webpage component nor the Jupyter Server can directly access each other to make Jupyter Notebooks accessbile online. The Apache web server is needed to facilitate communication between the components through the implementation of the reverse proxy shown in Appendix A.1. All components in the stack have a specific functionality and must work in sync with one another to achieve the overall business function [93].

However, this leads to tight coupling as discussed in Sections 2.3.2 & 3.2.1. In JAPyL, both the webpage and Jupyter components are highly dependent on the Apache web server for communication and marshalling of data to each other. If the operations of the Apache component is interrupted, communication between the other two components will be interrupted and can lead to catastrophic failure as discussed in Section 3.1.

5.2.2 Tier Elimination

To improve interoperation in distributed multi-tier software stacks, a proposed solution is to eliminate a tier and violate the principle of modularity discussed in Section 5.2. The number of interoperating components and tiers will be reduced and functionality of the eliminated component/tier will then be integrated within another existing tier [124].

Tier elimination leads to some integration of the functionalities between the client and server. Some of the server functionalities like Business Logic processing are embedded within the client in the Presentation Layer. However, there is still a distinct server side for functionalities like Data Access Layer as discussed in Section 5.3. This creates a new variation of the n-tier architecture shown in Figure 5.1.

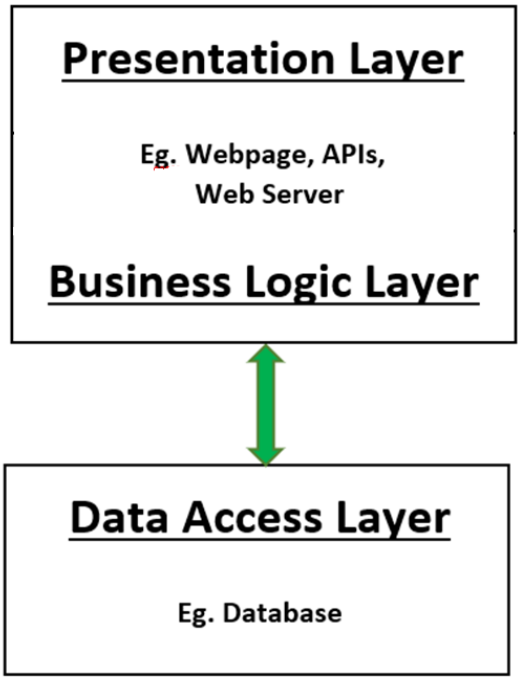

Figure 5.1: n-Tier Architecture Variation

Each tier in the new n-tier variation can still be hosted using either a single node (single tier distributed architecture) or two nodes (two tier distributed architecture) [93]. However, it is not possible to host each tier in its own node like the n-tier architecture as discussed in Section 1.1.2. The reason is that the Business Logic Layer is now integrated into the Presentation Layer [124].

5.3 Case Study

5.3.1 Tier Elimination in JPL

To investigate the tier elimination approach, a Jupyter, PHP & Linux (JPL) web stack is implemented shown in Figure 5.2. It eliminates the Apache component and carefully replicates the web services and security mechanisms of JAPyL within the webpage component using PHP libraries shown in Figure 5.2. Its architecture is based on the modified version of the n-tier architecture shown in Figure 5.1 and discussed in Section 5.2.2.

This approach is similar to JPyL where the Apache component is replaced with Flask microservices shown in Figures 3.2 & 3.3. However, JPL goes one step further. Instead of just

replacing the Apache component with PHP libraries, the web tier is actually eliminated and integrated within the webpage component shown in Figure 5.2.

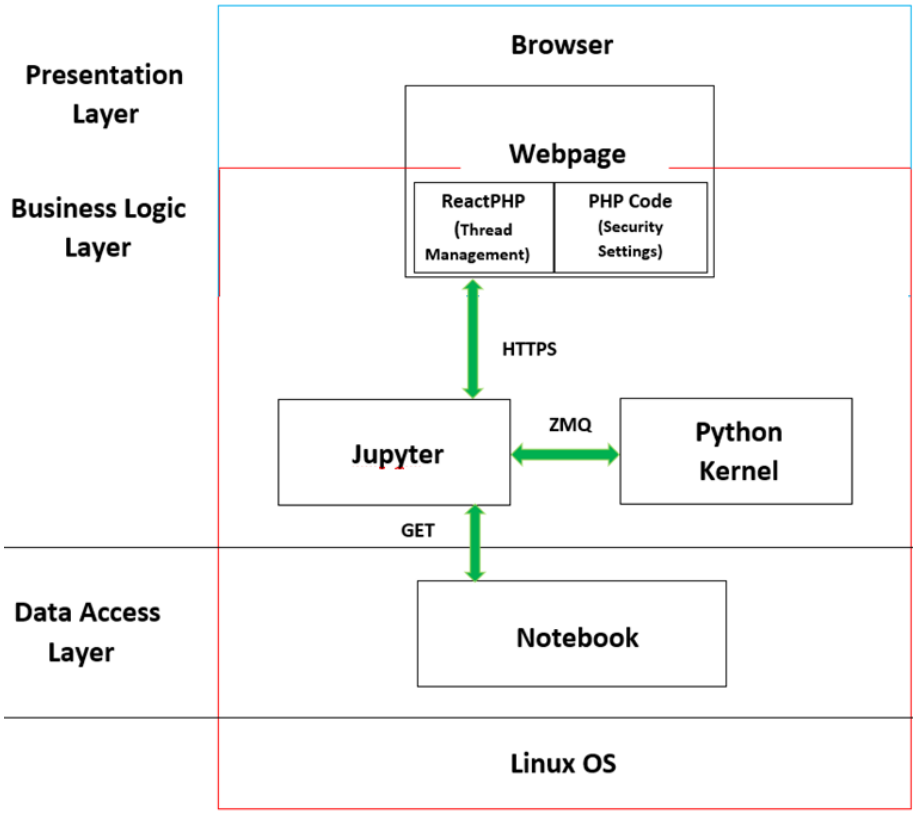

Figure 5.2: 3-Tier JPL Architecture

5.3.2 Implementing Server Requests in the Webpage

Usually in web stacks like JAPyL, the purpose of the webpage component is to act as the client interface for user interaction. However, in JPL, this component now has multiple functionalities. It must now display Jupyter Notebook content, accept client side requests, act as a reverse proxy, process business logic requests and marshall data to and from the Jupyter server [124].

There is a slight blurring of the boundary between the client and server tiers. However, there is still a distinct server side tier present. The Jupyter Server is still a separate component processing server side logic [124].

Server request processing in the webpage component occurs through the use of ReactPHP. This is a low level library that adds features to the PHP programming language like stream abstractions, async DNS resolver and HTTP/HTTPS client/server interaction processes [124]. Listing 5.1 shows a code snippet of how the webpage component processes HTTPS requests using ReactPHP.

Listing 5.1: Processing HTTPS Requests via the Webpage Component in JPL

1

```
2  <?php
3
4  $socket = new
       \React\Socket\Server('tls://192.168.43.135:10125', $loop,
       array(
5          'tls' => array(
6                  'local_cert' => 'servercert.crt',
7                  'local_pk' => 'mykey.key',
8                  'verify_peer' => false,
9                  'verify_peer_name' => false
10         )
11 ));
12
13 $server->listen($socket);
14
15 echo 'Listening on ' . str_replace('tls:', 'https:',
       $socket->getAddress()) . PHP_EOL;
16 $loop->run();
17
18 ?>
```

5.3.3 Additional Functionalities

There are some functionalities that are not available in the ReactPHP library especially security features like whitelisting/blacklisting that are provided by the Apache component shown in Figure 3.2. These features are hand-coded into the webpage component using PHP shown in the code snippet in Listing 5.2.

Listing 5.2: Blacklisting IPs via the Webpage Component in JPL

```
1
2  <?php
3
4  $deny = array("192.168.43.20", "192.168.43.60",
       "192.168.43.75");
5  if (in array ($_SERVER['REMOTE_ADDR'], $deny)) {
6     header("location: https://192.168.43.125:10125/");
7     exit();
8
9  }
```

```
10
11 ?>
```

5.3.4 Why PHP?

The rationale for using PHP to eliminate a web component is twofold. (1) It is one of the most popular and mature web programming technologies. (2) It offers a range of technical benefits [23, 147], as follows:

(1) PHP provides simple parsing and marshalling, e.g. parsing JSON and XML with a single line of code.

(2) PHP supports multiple major databases including MySQL, dBase, IBM DB2, InterBase, FrontBase, ODBC, PostgreSQL, SQLite, etc.

(3) PHP supports a range of frameworks like CakePHP, CodeIgniter, Zend, Larvarel that not only make development faster but also provide flexible coding styles and interfaces for programmers.

(4) PHP performance as a server side scripting language is comparable to C and exceeds Java.

For this study, the most significant benefit of PHP is that it features almost similar syntax and modules as Python shown in Table 5.1. Both languages have been combined into a new language called PyHyp as discussed in Section 2.4. Languages that are very similar interoperate with less semantic friction [11].

Feature	PHP	Python
Class Creation	Yes	Yes
Encapsulation	Yes	Yes
Functions	Yes	Yes
Immutable Data	Yes	Yes
Inheritance	Yes	Yes
Object Creation	Yes	Yes
Polymorphism	Yes	Yes
Shared State	Yes	Yes
Shared Memory	Yes	Yes

Table 5.1: Feature Similarities Between PHP & Python

5.4 Evaluation

5.4.1 Experiment Design

To evaluate the impact of tier elimination, the programmability and performance of JPL is compared with that of JAPyL on two platforms: Docker and the native RaspberryPi 3. While Jupyter stacks are mostly deployed in virtualized environments like Docker [172], it is far easier to obtain accurate core and memory resource measurements on the Raspberry Pi. Where possible, the components used in the JAPyL and JPL stacks are identical as follows: Docker 18.09, Raspbian Stretch, Jupyter Server 5.7.8, Ubuntu 16.04. JAPyL uses Apache Server 2.4.34, and JPL ReactPHP 0.8.4. Experiments for the Docker platforms are conducted on an Intel Core i3 system, Windows 10 Operating System with 2.23GHz and 8GB of RAM.

Two existing Jupyter Notebooks not written by the authors are downloaded from GitHub. They are selected to be well designed [132], i.e. to have a notebook title & introduction, descriptions of the model parameters, and of the data parameters, and to import packages. The notebooks are simply loaded by the JAPyL and JPL stacks.

To minimise variability, the reported results are the median of three consecutive benchmark executions.

5.4.2 Programmability

Code Size Code size is widely recognised as a measure of developer effort and of maintainability as discussed in Section 4.6.1. Table 5.2 enumerates the lines of code required to implement the functionalities of the JAPyL and JPL stacks. Implementing the JPL web stack requires 267 fewer lines of code, or 42% less code. This is to be expected because there are fewer tiers and languages to be implemented (Table 5.3).

Table 5.2: Lines of Code (LOC)

Functionality	JAPyL	JPL
Embed Notebook	10	36
Host Webpage	29	26
Reverse Proxy	33	23
Security Configs	21	31
Language Processing	527	63
Messaging	11	85
Total	631	364

Language Implementation Many distributed multi-tier software stacks consist of a polyglot languages as discussed in Sections 2.3.2, 2.4 & 4.6.1. However, the main reason for a smaller

Table 5.3: Programming Language Implementation

Functionality	JAPyL	JPL
Embed Notebook	HTML	HTML
Host Webpage	Apache Configs	PHP
Reverse Proxy	Apache Configs	PHP
Security Settings	Apache Configs	PHP
Language Processing	Python	PHP
Messaging	JSON	PHP
Total	4	2

code size in JPL is that it utilises fewer programming languages and paradigms. JAPyL requires four languages for complete implementation whereas JPL requires only two shown in Table 5.3.

Code Complexity With fewer tiers the JPL developer writes less code, but how complex is the code? This is dependent on how many paradigms the programmer must use and how many control flows must be managed [137].

The expectation is that since JPL uses fewer programming languages than JAPyL in implementation, the code complexity will be less. However, the evaluation reveals that the structural code complexity needed to interoperate the components and PHP language in JPL is slightly higher when compared to JAPyL based on Cyclomatic Complexity shown in Table 5.4.

Table 5.4: Cyclomatic Complexity (cc)

Functionality	JAPyL	JPL
Embed Notebook	1	1
Host Webpage	1	1
Reverse Proxy	1	1
Security Configs	1	5
Language Processing	13	1
Messaging	2	12
Total	19	21

A higher complexity number means that the programmer may have to deal with more control paths in the JPL code. Higher complexity leads to a decrease in abstraction because more programming effort is now required to achieve a particular functionality in the system [22].

5.4.3 Abstraction

As discussed in Section 5.3.3, both stacks feature a Defense-in-Depth multi-layered security approach (Figure 2.4). That is, various security mechanisms are deployed throughout the stack. The intention is that if an attacker targets the Jupyter Notebook online and is able to penetrate one layer, another layer may thwart the attack as discussed in Section 2.1.3.

In JAPyL, the security mechanisms are typically declaratively specified in either the Jupyter Server or Apache. Often security features are implemented as runtime parameters. Appendix B.1 provides examples of some of the declarative security mechanisms implemented in JAPyL.

It is commonly believed that such declarative specifications in some domain-specific language like HTML, XML, or Apache Configs makes things easier for the developer by raising the level of abstraction [85]. However the developer must be fluent in a range of languages shown in Table 5.2. In JAPyL these are (1) Jupyter Configurations (2) Jupyter Notebook Interoperability Techniques i.e. Kernel, Subkernel & Magic Implementations (3) Several different languages including JSON, Apache Configs, etc [99].

In contrast, much of the security implementations for JPL are imperative and hand coded as shown in Table 5.5. That is, only PHP was written to perform the necessary configurations as demonstrated by the examples in Table 5.5.

Table 5.5: JAPyL vs JPL Security Implementations

Functionality	JAPyL	JPL
Password Encryption	Command Line	Command Line
Port Spoofing	Hand Coded	Hand Coded
SSL Encryption	Apache Configs	Hand Coded
IP Whitelisting	Apache Configs	Hand Coded
IP Blacklisting	Apache Configs	Hand Coded
Security Headers	Apache Configs	Automated

Using an imperative paradigm means that the level of abstraction has decreased compared with JAPyL. However the developer has the expressiveness to implement the necessary functionalities, and can potentially implement features not supported by the JAPyL DSLs [40].

5.4.4 Performance: Latency and Throughput

Given the reduced number of tiers in JPL compared with JAPyL and that PHP is similar to Python as shown in Table 5.1, it is reasonable to expect JPL to outperform JAPyL. However, this is not the case.

Figures 5.3 & 5.4 show the request latencies of JAPyL and JPL as the number of concurrent connections varies from 100 to 1000 on all platforms. Contrary to expectations both figures show that JPL latency is nearly two or three times greater than JAPyL.

Figures 5.5 & 5.6 show the request throughput of JAPyL and JPL for 100 concurrent users. It confounds expectations by revealing that JPL throughput is at least one to two orders of magnitude lower than for JAPyL. This is a result of the relatively poor thread management in the PHP React library.

Realistically, despite the advances made by PHP in terms of concurrency, the language itself is still naturally not quite oriented towards multi-threading. The ReactPHP library sup-

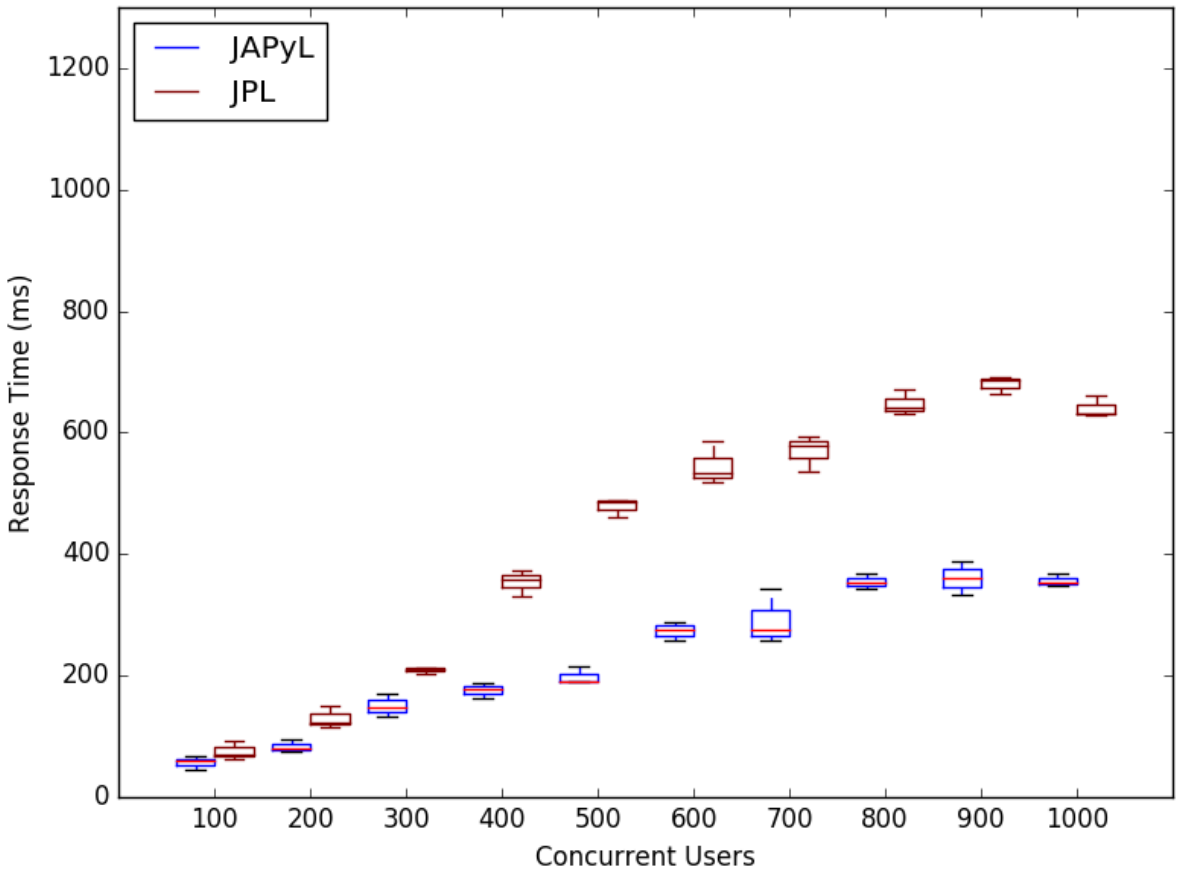

Figure 5.3: JAPyL vs JPL Latency (RaspberryPi)

ports multi-threading to some degree by allowing a limited number of non-blocking I/O operations [124]. However, this cannot compare to Apache that has several sub-processes running in parallel one for each request, up to a couple of dozens or hundreds, depending on the configuration [78]. Simply, Apache is built for concurrency, parallelism and to utilise multicore systems.

5.4.5 Resource Usage: Core Utilization, Memory Overhead

As the JPL stack eliminates Apache and runs fewer components it is reasonable to expect JPL to consume less resource than JAPyL. Figure 5.7 shows the CPU utilisation of JAPyL and JPL as they process HTTPS requests with 100 concurrent connections on the Raspberry Pi platform. Again, contrary to expectation JPL utilisation is typically 20% greater than JAPyL.

A possible explanation for this is memory leaks. This refers to a situation where memory is not deallocated after program execution [44]. The amount of memory utilised by the application will continue to slowly increase over time. In PHP, there is some that just cannot be freed up on a regular basis due to its reliance on reference counting to manage memory [80].

Garbage collection is utilised to pick up the pieces that the reference counter misses. How-

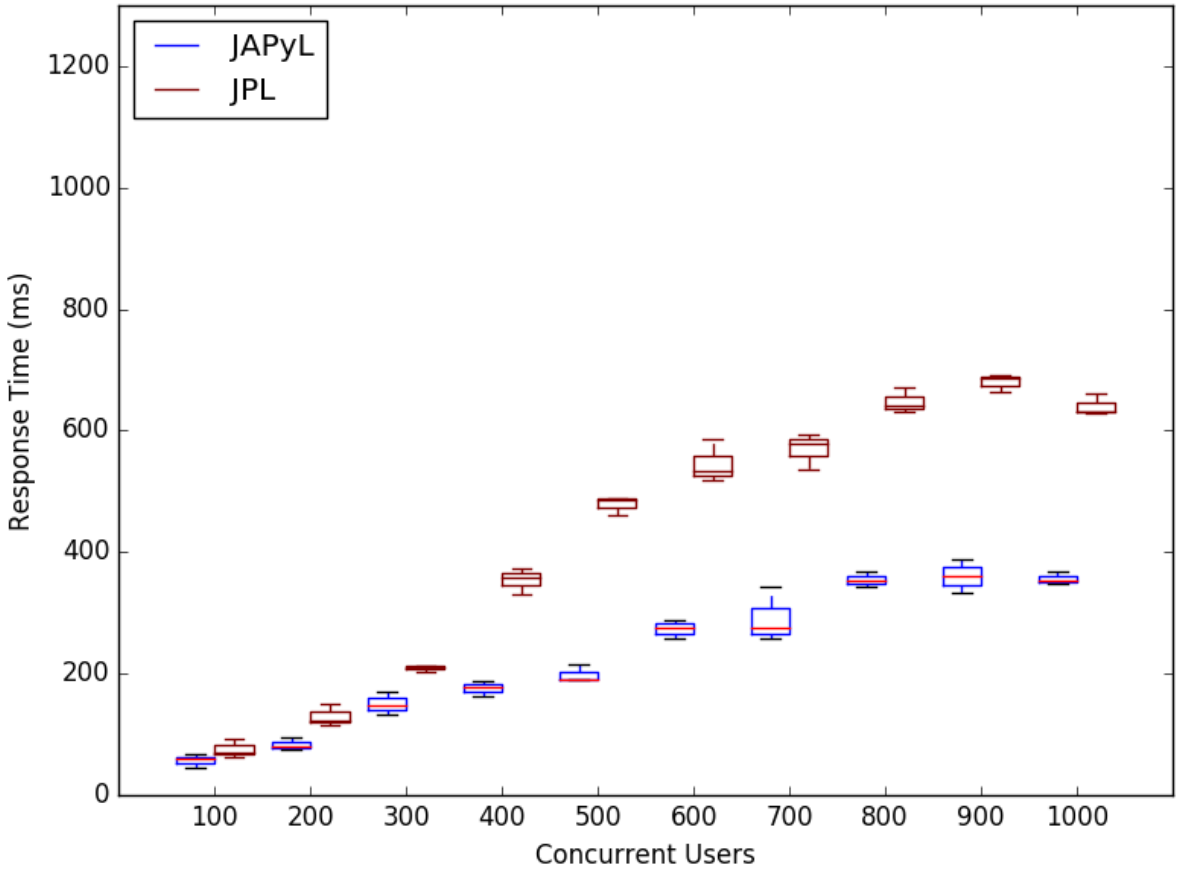

Figure 5.4: JAPyL vs JPL Latency (Docker)

ever, in PHP, several factors can prevent the efficient freeing up of memory resources. For example, if an object's refcount increases and never decreases, then the object is still technically in use and is not garbage. The garbage collector, therefore, cannot identify it as such and will not free up the object [147].

5.5 Evaluation Summary & Reflection

Summary This evaluation explores whether reducing the number of tiers/components improves interoperability. This is done by systematically comparing the 4-tier JAPyL and 3-tier JPL web stacks (Section 5.3). The key findings from our case study are as follows.

Through the elimination of the Apache component, less developer effort is required. JPL requires 42% less lines of code (Table 5.2) and fewer languages for implementation (Table 5.3). Furthermore, the level of abstraction at the upper levels of the stack has decreased in JPL but the developer is able to automate certain functionalities like security headers (5.4.3).

However, the complexity (Table 5.4) for implementation is greater. The primary reason is that there is more handwritten code in JPL, e.g. for security (Section 5.3). This decreases the

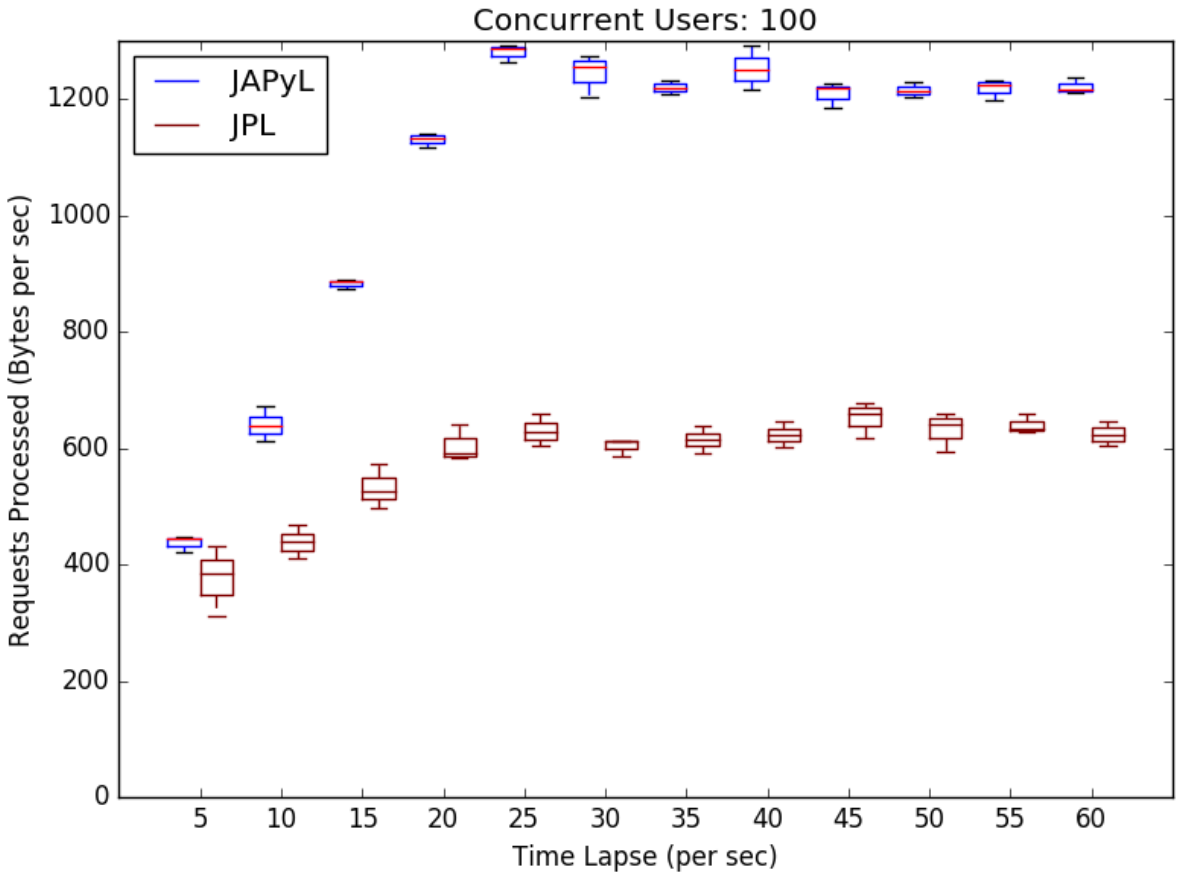

Figure 5.5: JAPyL vs JPL Throughput (RaspberryPi)

level of abstraction in JPL when compared to JAPyL.

Eliminating the Apache component, and associated interoperation, in JPL and replacing it with a PHP threaded library increases latency (Figures 5.3 & 5.4) and reduces throughput (Figure 5.5 & 5.6). This reflects that the Apache thread management is far superior to that provided by the PHP React library (Section 5.4.4).

Despite replacing Apache, JPL consumes more resources. JPL uses 20% more cpu resources (Figures 5.7). It is possible that this is a result of memory leaks in PHP.

Generalisation Based on the evaluation, the tier elimination technique in a multi-tier software stack will improve interoperation to some degree. Similar to the evaluation of tierless iTasks/mTasks in Chapter 4, there will be a reduction in the number of polyglot languages required for implementation. This can have significant benefits. There will be (1) Reduced developer effort as less source lines of code (SLOC) are required for implementation of the stack. (2) Less programming languages and paradigms will lead to less chances of semantic friction occurring that improves reliability. (3) Programmers have the ability to automate certain functionalities at the upper levels of the stack due to decreased abstraction. However, the issues or performance, resource consumption and code complexity are even more profound.

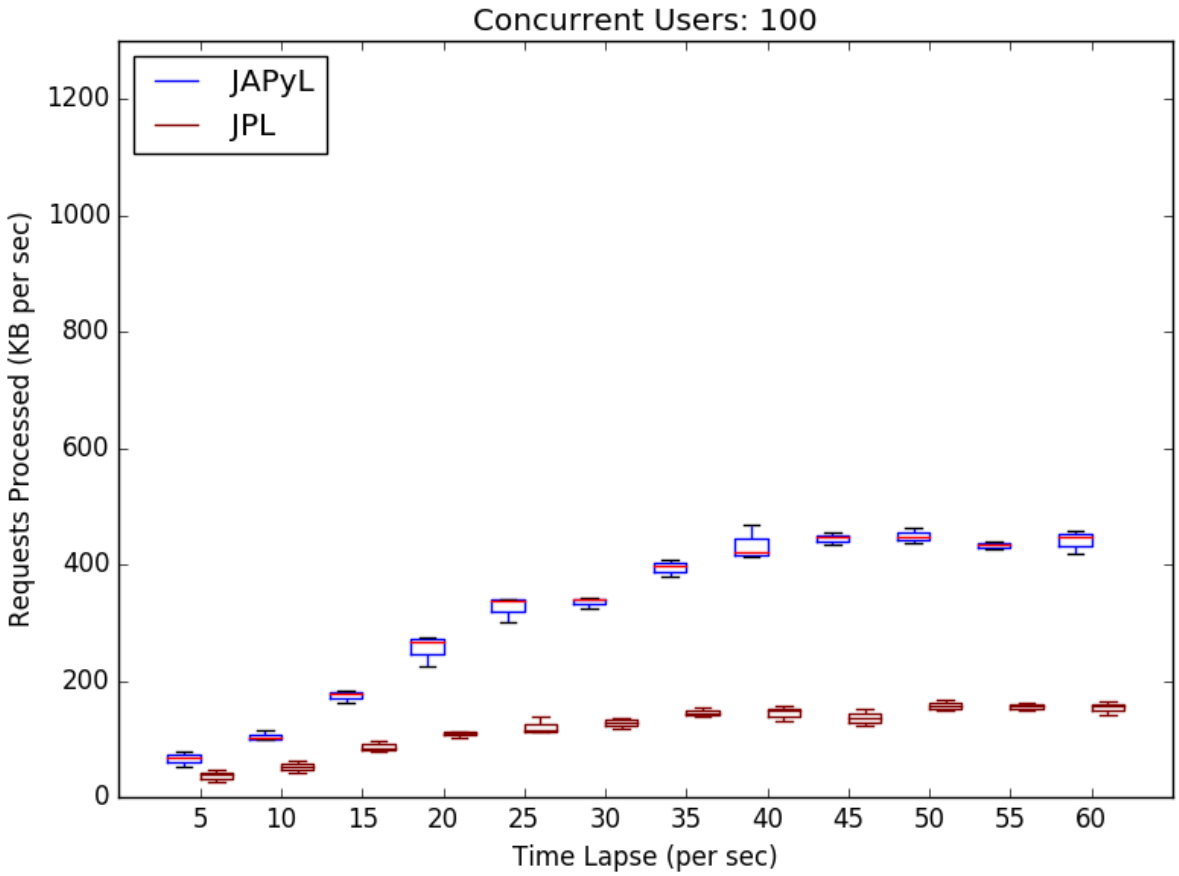

Figure 5.6: JAPyL vs JPL Throughput (Docker)

Reflection It is possible that substituting Apache with PHP is a poor decision. Using a language with better memory management and support for multi-threading, like Erlang or Go, could possibly better meet these expectations.

Furthermore, just because a system or application has fewer components or layers does not usually make it simpler or less complex. You still have to take into account the processes and behaviour interactions that may be impacted as well as the compatibility factor between system components and languages.

5.6 Chapter Summary

This chapter explores whether eliminating a tier/component improves interoperability in distributed multi-tier software stacks. This is accomplished through eliminating a tier/component and integrating its functionality into another existing tier.

To illustrate this technique, the Apache web component in 4-tier JAPyL is eliminated. The functionalities of this component are then carefully replicated utilising the ReactPHP code library within the webpage component to form 3-tier JPL. This results in the webpage component

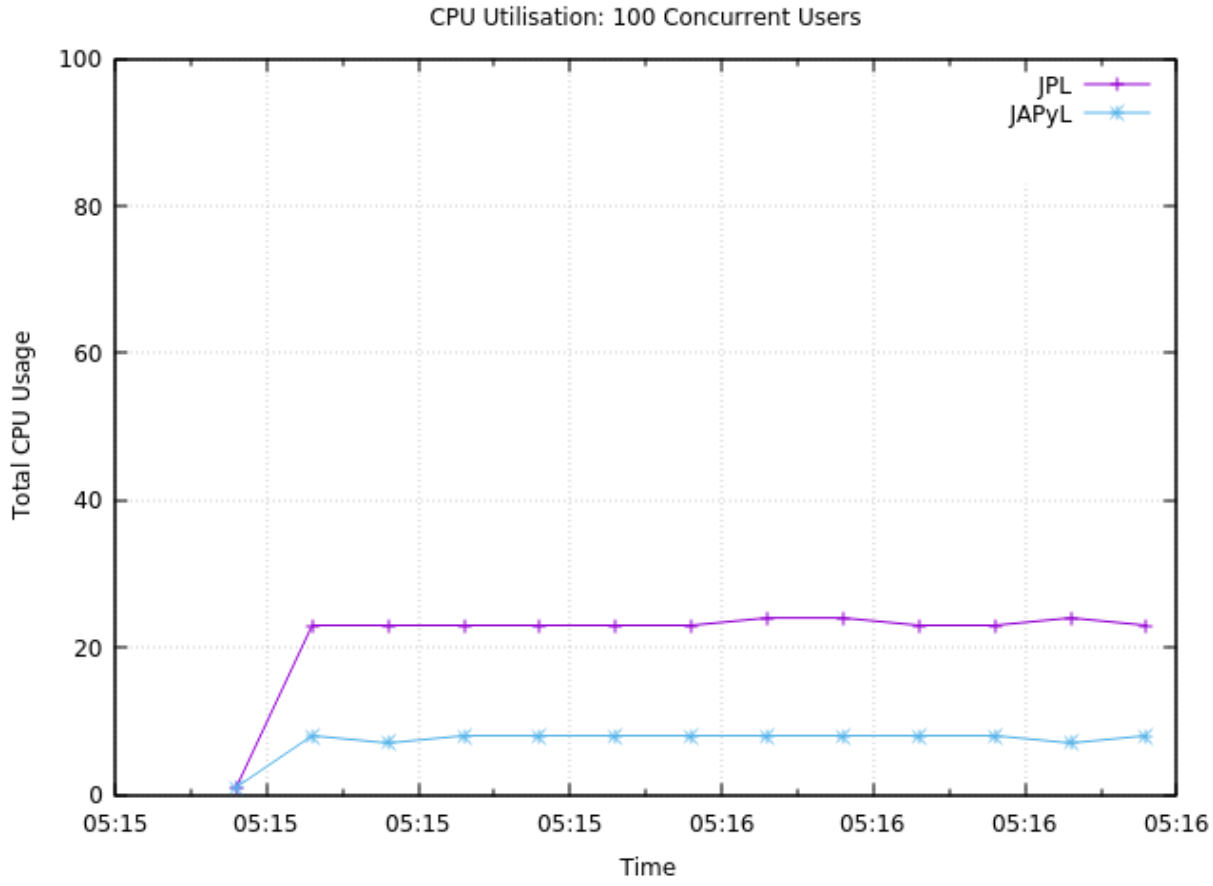

Figure 5.7: JAPyL vs JPL CPU Resource Utilisation (RaspberryPi)

now having to perform both client side and some server side functionalities.

Evaluations show that eliminating a tier within JPL does improve interoperability to some degree. There is a reduction in the number of programming languages and paradigms utilised that results in a smaller code size when compared to JAPyL. This can lead to improved reliability as there is less semantic friction. However, the issues of performance, resource consumption and code complexity are increased. These issues may be a result of the poor decision to use PHP threaded libraries as the programming language of choice.

Chapter 6

Conclusion

The aim of this thesis is to investigate techniques to improve heterogeneous interoperability in multi-tier software stacks. It investigates common approaches like microservices and tierless languages. These techniques are shown to improve interoperation to some degree but challenges still remain. Thus, an alternative approach involving the elimination of a tier/component is evaluated as a potential solution to solving the challenges of heterogeneous interoperability.

6.1 Summary

Chapter 2 provides background information on the different forms of interoperability and how they relate to heterogeneous interoperability in distributed multi-tier software stacks (Section 2.3). It is shown that the interoperation challenges are similar for both Web & IoT software stacks and that many of the problems are attributed to tight coupling and multi-language integration (Section 2.3.2).

These two factors can lead to several interoperability challenges including the following:

(1) Interoperating components in multiple languages and paradigms increases developer cognitive load since they must simultaneously reason in multiple languages and paradigms.

(2) There must be correct interoperation of components, e.g. adherence to the API or communication protocols between components.

(3) Interoperation between different components can lead to diverse modes of failure as each component can fail in unique ways.

Several attempts have been made to improve interoperation in distributed multi-tier software stack. These include the utilisation of techniques like reducing the number of languages implemented in distributed software stacks (Section 2.4.1), introducing notebook programming

(Section 2.1.2), browser compilation (Section 2.4.1), combining programming languages (Section 2.4.1) microservices implementation (Section 2.4.2) and using tierless languages (Section 2.4.3). Each technique improves interoperability to some degree but does not solve all interoperation challenges.

Chapter 3 investigates whether microservices are able to improve interoperation as they provide *improved reliability* due to the loose coupling of services (Section 3.3). Some simple experiments show that this notion is accurate. Component based stacks like JAPyL and WordPress fail catastrophically. In contrast, the individual microservices in the JPyL microlith architecture prevents catastrophic failure (Section 3.3.2).

Microservices are commonly classified based on the properties of dependence or state (Section 3.6). A binary reliability classification is added and combined with the other classifications to define a three dimensional space: the MDSR Classification in Figure 3.12 (Section 3.8). Using three established web applications (Section 3.5), it is shown that microservices exemplify the six MDSR classes (Tables 3.4 & 3.5). A prototype static analyser that can statically identify all six classes in Flask web applications is outlined and applied to seven small web applications (Section 3.8.6). Analysing the applications reveals that the majority of services are chained (70%), stateful (73%) and critical (77%) (Table 3.6). It is speculated that the high percentage of critical services indicates that these applications are not designed for reliability.

The reliability implications of the different MDSR classes were evaluated by running the case study applications against a fault injector to show the following. (1) All applications fail catastrophically if a critical microservice fails (Section 3.9.2). (2) Applications survive the failure of individual minor microservice(s) (Section 3.9.3. (3) The failure of any chain of microservices in JPyL & Hipster is catastrophic (Section 3.9.5). (4) Individual microservices do not necessarily have minor reliability implications (Section 3.9.5).

Chapter 4 reports the first comparison of interoperation in tiered and tierless IoT stacks through the systematic evaluation of four implementations of the prototype UoG smart campus IoT system: two tierless (CRS & CWS) and two Python-based tiered (PRS & PWS). It shows that tierless languages offer real benefits for IoT development.

Analysis of the implemented code base reveals the following (Section 4.6.1). (1) Tierless developers manage less interoperation. (2) Tierless developers benefit from automatically generated, and hence correct, communication. (3) Tierless developers can exploit powerful high-level declarative and task-oriented IoT programming abstractions. Indeed, tierless developers do manage less interoperation: CRS uses a single DSL and paradigm, and CWS uses 2 DSLs in a single paradigm and 3 source code files. In contrast, both PRS and PWS use at least 6 languages in 2 paradigms and spread over at least 35 source code files (Tables 4.2 and 4.3). Also, developers write a sixth of the amount of communication code (Figure 4.9). The added benefit

of reduced developer effort, code and paradigms for interoperation is that there is less semantic friction that will improve reliability (Section 4.6.6).

However, there are some challenges for interoperation with tierless languages especially in IoT stacks. Maintainability efforts in a tiered and tierless approach are similar (Section 4.6.7) while there is more community support for mainstream tier based programming languages (Section 4.6.8). Furthermore, tierless programmers must master new tierless programming abstractions, and the semantics of these automatic multi-tier behaviours are necessarily relatively complex.

Chapter 5 explores a pragmatic approach to reducing interoperation in distributed multi-tier software stacks, namely eliminating a tier/component. It reports the implications of eliminating the Apache web server in a JAPyL web stack: Jupyter Notebook, Apache, Python, Linux, and replacing it with PHP libraries to form a new stack referred to as JPL (Section 5.3).

It shows that eliminating the Apache component, and associated interoperation, in JPL and replacing it with a PHP threaded library increases latency (Figures 5.4 & 5.3) and reduces throughput (Figures 5.6 & 5.5). This is to be expected since Apache thread management optimization is far superior to that provided by the PHP React library (Section 5.4.4). Furthermore, despite replacing Apache, JPL consumes more resources. JPL uses 30% more core cycles (Figure 5.7).

In terms of programmability, the structural code complexity needed to interoperate the components and PHP language in JPL is slightly higher when compared to JAPyL based on Cyclomatic Complexity. Table 5.4 shows that JAPyL has a rating of 19 when compared to 21 in JPL. This means that the code in JPL has more control flows that have to be managed. There is also a decrease in abstraction at the higher levels of the stack due to the removal of the Apache component (Section 5.4.3).

However, implementing the JPL web stack requires 267 fewer lines of code, or 42% less code (Section 5.4.2). This is to be expected because there are fewer languages and paradigms to be implemented as shown in Table 2.1. JAPyL utilises 3 programming languages compared to 2 in JPL. Moreover, even though the abstraction levels have decreased in the upper levels of the stack in JPL, hand-written PHP code in JPL can provide additional capabilities, e.g. automating the handling of security headers (Section 5.4.3). Thus, interoperation is improved to some degree because there is less developer effort for stack implementation and there is reduced chances of semantic friction that improves reliability.

6.2 Tradeoffs/Limitations

6.2.1 Interoperability

Maintaining correct and secure interoperation in distributed multi-tier software stacks remains a significant challenge. In 2004, after integrating and upgrading components, PayPal experienced periodic outages for almost five consecutive days. One of the major reasons for this was a result of integration errors and incompatibility of some third party components [84].

In 2013, Spotify online platform suffered from a cascade of failures while Twilio billing service platform malfunctioned because it kept on repeatedly billing customers for ten consecutive hours. Even though the service failures for both platforms were different, both were a result of a malfunctioning chained microservice that affected other services [60].

Telecom company TalkTalk suffered a major breach attack in 2015 that resulted in personal information of its customers being stolen. The breach was a result of a database component being vulnerable to a SQL Injection attack that occurred through one of the Webpage components of the company's platform [118].

Unfortunately interoperability challenges are only likely to get worse as applications in the Web & IoT domains are becoming even more complex. There is an ever increasing demand for more interactive features, collaboration between different clients, communication with cloud technologies and support for offline functionalities [114]. Developers have to find innovative techniques to ensure seamless interoperation of the different tiers in these distributed multi-tier software stacks. For example, uMiddle is a distributed middleware framework that utilises mappers, translators and agnostic platform protocols for components from different applications to communicate with one another [15]. However, this increasing demand is making the challenges of interoperation in distributed multi-tier software stacks even more complex.

Heterogeneity A key requirement of many distributed multi-tier software stacks is that different components must interoperate (Section 1.2). Now, this challenge is being multiplied as the stacks become even more complex. For example, ensuring that IoT applications can communicate with cloud technologies typically entails combining different remote method invocation protocols, messaging protocols and software bridges [16].

Performance Heterogeneity will continue to impact the performance of the stack. Interprocess communication is typically slower than a function call as data must be marshalled for communication between languages and components [11, 82]. This trend continues even in more modern techniques for improving interoperability. For example, the Starlink framework generates runtime interoperability logic to create agnostic protocols to ensure communication between different components on different platforms [16]. However, there is a performance overhead. It takes six times longer for data to be marshalled and understood by different components when

compared to systems that interoperate using similar component specs [16].

Security Exposing APIs for interoperation makes stacks vulnerable to security attacks. For example, the components in web stacks like LAMP have left web applications vulnerable to attacks like SQL Injection, Cross Site Scripting (XSS) and Clickjacking. These are usually result of a combination of unpatched servers, poor coding habits or a lack of proper validation or sanitization in frontend user components [74].

This makes securing component based IoT stacks even more challenging. The increasing complexity of the stack multiplies the number of security challenges. Security architects have to categorise vulnerabilities based on each tier of the stack as well as consider the characteristics of a particular IoT environment. For example, they may have to consider what type of sensors are being used or what type of protocols are being implemented for a particular application [76].

Team Development This research focused on how interoperability affects the full stack developer (Section 2.3.2). However, in large organisations, there are teams of developers that implement and maintain large scale distributed multi-tier software stacks. Distinct development teams tend to implement different tiers or components in parallel [98]. There are many studies in the literature that focus on how team development affects interoperability and influences the use of certain interoperability techniques. For example, at Netflix, cross functional teams are created to implement and maintain each individual microservice in a platform [98]. The tradeoffs of such an approach is beyond the scope of this research.

6.2.2 Microservices

Microservices architectures are based on the idea of self containment. Every service is loosely coupled and independently deployable as discussed in Section 2.4.2. However, Chapter 3 shows that there are different types of microservices like chained microservices and that some are poorly implemented and prone to catastrophic failure.

Even if the microservices architecture is implemented with the appropriate principles and patterns the technique has certain limitations. There are some microservices that remain critical to the core functionality of the system. For example, if a service that handles credit card payments in an e-commerce fails, the acceptance of payments for the business is not possible. In an IoT application, if a service that handles the transfer of data from the sensors to the data store fails, the application cannot collect data. Hence we find that even individual microservices may be critical as discussed in Section 3.8.5.

Additionally, developers tend to implement an increasing number of microservices in an application as it expands [2]. This is not always a good practice as it can lead to more critical as well as chained microservices. There are some studies that suggest that increasing the number of microservices in an application will make it more fault prone like [95]. Table 3.3 refers

to this practice as the Microservices Greedy bad smell. However, increasing the number of microservices in an application does improve scalability.

New instances of a service can be launched to split the load on the system to enhance performance [43]. There are, however, limits. For example, one study by [43] shows that gradually increasing the number of instances of a microservice in the cloud improves the throughput of the application over time. However, after over one hundred instances of the microservice has been implemented, throughput no longer improves. This is referred to as microservices tails-of-scale [43].

6.2.3 Tierless Languages

Tierless languages are an emerging technique especially in the IoT domain to reduce interoperation complexity. This research extensively shows the benefits and potential of tierless languages in the IoT domain. There is further evidence with established languages like Erlang or Elixir being utilised to engineer IoT systems in production environments as discussed in Section 2.4.3. However, tierless languages are not a panacea.

While offering real benefits for IoT systems development, tierless languages also raise some challenges. Programmers must master new tierless programming abstractions, and the semantics of these automatic multi-tier behaviours are necessarily relatively complex. In the Clean context this entails becoming proficient with the iTask and mTask DSLs. Moreover, specifying a behaviour that is not already provided by the tierless language requires either a workaround, or extending a DSL. However, implementing the relatively simple smart campus application required no such adaption. Finally, tierless IoT technology is very new, and both tool and community support have yet to mature [88].

Tierless languages often require the development of novel and esoteric domain specific languages (DSLs) and Chapter 4 uses iTasks and mTasks DSLs. They are both written in DSLs embedded in Clean, a fairly stable industrial-grade but niche functional programming language [88].

Currently many tierless languages are research languages. In such languages there is limited documentation, training materials, and a very small team developing/maintaining the language [88]. Of the technologies used in Chapter 4, while iTasks is industrial grade and far better documented and maintained, mTasks is a research DSL, so lacks the necessary documentation and maintenance [88].

Additionally, tool support for many of these languages is lacking. For example, there is a Clean IDE tool for debugging Clean programming code. However, the tool does not support iTasks/mTasks as discussed in Section 4.6.8. Developers of tierless languages usually have to learn by experience in debugging and testing programs developed with these sophisticated DSLs [88].

Another major issue is the technical challenge of specifying an application in a tierless lan-

guage. Developers are required to meticulously annotate code to support tier splitting and this can become time consuming and error prone. For example, during compilation, tier splitting requires the tier specific code to be modular [164]. This is not supported by all languages as in the case of Ur/Web. It does not support separate compilation of modules. The language relies on whole-program analysis for splitting the application into client and server programs [164].

There is then the significant limitation of tierless languages being applied mostly to systems with the distributed client server architecture. There is a general lack of support for other generic distributed architectures like Peer-to-Peer (P2P) or Data Centred architectures [164]. One exception is the ScalaLoci language that has modules to support the P2P architecture. Future tierless languages may address this limitation.

6.2.4 Tier Elimination

Tier Elimination remains experimental. A more substantial study could determine how effective this technique is for improving interoperability. Moreover, tier elimination violates the established principle of modularity as discussed in Section 5.2.

This technique also violates the accepted notion of abstraction in distributed multi-tier software stacks as discussed in Section 5.4.3. In tiered multi-tier software stacks, it is generally accepted that the upper layers of the stack will have increased levels of abstraction while the lower levels of the stack will have decreased levels of abstraction [88].

Clean mTasks goes one step further and attempts to increase the levels of abstraction at the lower levels of the IoT stack as discussed in Section 4.6.5. However, this is in contrast to Tier Elimination. This technique decreases the levels of abstraction at the upper levels of the web stack as discussed in Section 5.4.3.

In the absence of substantial studies it is difficult to make informed speculation as to all of the tradeoffs of tier elimination. However, decreasing the levels of abstraction does increase the likelihood of more bugs being introduced into a system.

In using an imperative paradigm to implement the stack like in the case of JPL in Section 5.3, the developer has to describe all the steps necessary to perform a computational task. Specifying every task manually rather than using automation through the use of a declarative paradigm is generally considered unsafe as it can lead to poor coding implementations or the developer making mistakes that introduces bugs [115].

6.2.5 Research Limitations

Finally, this research suggests that there is no single solution that can solve all the challenges associated with heterogeneous interoperability in distributed multi-tier software stacks. However, the evaluations and results presented have certain limitations.

Small Case Studies Many of the evaluations and results are based on relatively small case studies. For example, the MDSR work is based on web applications that consist of no more than ten microservices. In the real world, some web applications are implemented with hundreds of microservices like the Titus platform from Netflix. Similarly, the work involving the tiered vs tierless smart campus case studies is based on small IoT systems i.e. 700 SLOC in 33 source files.

Metrics Some of the results especially in comparing different programming languages are based on the SLOC metric. SLOC is a common code size metric, and is especially useful for multi-paradigm systems like IoT systems. It is based on the simple principle that the more lines of code, the more developer effort and the increased likelihood of bugs [129]. It is a simple measure, not dependent on some formula, and can be automatically computed [136]. However, SLOC must be used carefully as it is easily influenced by programming style, language paradigm, and counting method [3].

Another issue is that the SLOC metric only provides data at a particular point in time. For example, in Table 5.2, it is shown that the JPL web stack requires 267 fewer lines of code, or 42% less code (Section 5.4.2) when compared to JAPyL. Over time, it is expected that more complex functionalities will be required from both stacks. Given that JPL utilises an imperative paradigm, it could be argued that more complex functionalities will eventually lead to JPL requiring more lines of code when compared to JAPyL that uses a declarative paradigm. There is already some evidence for this. Table 5.4 already shows that JPL code is more complex when compared to JAPyL. Thus, the SLOC metric is perhaps not a good metric to determine how much effort is required by a developer to implement future functionalities in a system.

Programming Languages Each programming language has its advantages and disadvantages for usage. Selection of a programming language is especially important in the building of any application. The use of PHP in the tier elimination work to build JPL may not have been the best decision as it could have possibly contributed to its poor latency, throughput and resource consumption that may have affected the validity of the results presented in the evaluation.

6.3 Future Outlook & Recommendations

6.3.1 Microservices

There is no doubt that the use of the microservices architecture in large-scale industrial Web/IoT applications to improve interoperability will continue. In the IoT domain, microservices are becoming more popular as a means of integrating tiered IoT applications with cloud based platforms like IBM Cloud and Siemen's MindSphere [2]. The loose coupling and independently deployable nature of microservices are supposed to almost guarantee the continuous publishing

and retrieval of data from the Perception Layer to the Presentation Layer without catastrophic failure [17].

This is again based on the assumption that all microservices deployed in multi-tier software stacks consist of the individual property. This research shows that this is not always the case. Table 3.6 reveals that over 70% of microservices implemented in open source web applications consist of the chained/stateful/critical property while only 17% consist of the individual/stateful/minor property. It reasonable to speculate that this trend is similar for microservices in IoT applications.

It is recommended that when implementing microservices in large scale Web/IoT applications that developers attempt to implement individual/stateful/minor or individual/stateless/minor services as far as possible. Section 3.7 confirms [60] in showing that any chained microservice is critical.

Moreover, developers must remember that individual microservices implementations may also be critical. Table 3.6 shows that 7% of microservices implemented are individual/stateful/-critical and 3% are individual/stateless/critical. Thus, despite the many perceived benefits in the academic and grey literature, individual microservices can cause catastrophic failure.

6.3.2 Tierless Languages

The future of tierless may involve the use of mainstream programming languages rather than DSLs especially in developing large scale industrial Web/IoT applications. The idea here is still to minimise the number of programming languages in a stack. One example is the use of a tierless JavaScript framework called Slicing Tierless JavaScript Programs (STIP.js). Here, the developer writes annotated code blocks of JavaScript together in one module to specify the tiers. The code is then compiled and sliced to automatically form the necessary client/server tiers [114].

It is recommended that a single mainstream programming language be used and compiled to synthesise the tiers/components in multi-tier software stacks. The main advantage of this technique is that the developer will have the benefit of being able to utilise a mainstream language that is well documented and has tool support especially for testing and debugging code.

This trend is readily becoming adopted in industrial applications. Google Web Toolkit (GWT) is a framework utilised in many Google applications like Google Adwords & Google Wallet. It enables developing an entire web application in Java. The GWT compiler facilitates the translating and splitting of code from Java to JavaScript [112].

6.3.3 Tier Elimination

It is still unclear whether Tier Elimination will become accepted as a technique to improve interoperability in large-scale Web/IoT industrial applications. However, it might have some

practical use.

Given that IoT multi-tier software stacks are even more complex than web stacks as discussed in Section 1.1.3, Tier Elimination has the potential to reduce the number of tiers/components at the Presentation & Application Layers. Eliminating a web server or component or even integrating some components in the Application Layer with the Presentation Layer can simplify the tiered IoT stack.

This might be especially useful for edge IoT applications. These are applications where client data is processed at the periphery of the network [19]. By doing this, users should benefit from faster data processing, analytics and services. However, Section 5.4.4 has shown that performance overhead remains a significant issue. This may have been the result of the choice of language. Thus, it is recommended that research using a faster processing language like Erlang may be necessary to really determine if Tier Elimination has the potential as a large-scale industrial technique to improve interoperability.

6.4 Future Work

Microservices Reliability A potential plan is to investigate larger microservice-based systems, e.g. Death Star[1], to accumulate evidence for the effectiveness of the MDSR Classification for identifying reliability bad smells. It would also be interesting to see whether MDSR could be extended to also classify failures in infrastructure services.

Furthermore, it may be possible to enhance the analyser to make it more comprehensive, e.g. to automatically detect more persistent stores. Perhaps the analyser could suggest when some microservice resilience patterns, like bulkhead or load balancer, could reduce criticality? Finally, it may be worthwhile to investigate the potential of combining static MDSR analysis with dynamic SDG analysis.

Tierless Languages The metrics reported like code size, numbers of source code files, and of paradigms are only indirect, although widely accepted, measures of development effort. A more convincing evaluation of tiered versus tierless technologies could be provided by conducting a carefully designed and substantial user study, e.g. using N-version programming.

Furthermore, a study that implemented common benchmarks or a case study in multiple tierless IoT languages would provide additional evidence for the benefits of the tierless approach over tiered systems in IoT stacks. This would further justify the demonstration and comparison of alternative design decisions like tierless languages.

Tier Elimination Further investigation may involve replacing stack components with more appropriate programming language technologies for a variety of stacks and domains. The inten-

[1] https://github.com/djmgit/DeathStar

tion is that tier elimination should have minimal impact on an application stack functionality, but instead seek to explore potential non-functional benefits. To this end, maybe developing an IoT stack using tier elimination and comparing it against a tierless Clean iTask/mTask implementation of a smart campus IoT stack could further validate these results.

Appendix A

Interoperability in JAPyL Web Stack

A.1 Interoperation between Apache & Jupyter Components

In JAPyL (Section 2.1.2), the Jupyter Notebook is embedded in the webpage via HTML using the IFRAME Tag. However, by default for security reasons, the Jupyter Notebook does not allow its Notebooks to be embedded externally. This feature is enabled in the Jupyter Server using Jupyter Configs in the jupyter_notebook_config.py configuration file.

Listing A.1: Jupyter Configuration for Enabling Notebook Embedding

```
1
2 c.NotebookApp.ip = '0.0.0.0'
3 c.NotebookApp.tornado_settings = {
4 'headers':{
5    'Content-Security-Policy': "frame-ancestors
       https://192.168.43.100:10125 'self' "
6  }
7 }
8 c.NotebookApp.base_url = '/jupyter/notebooks/'
9 c.NotebookApp.open_browser = False
10 c.NotebookApp.port = 9253
```

The Apache web server is then configured to act as a reverse proxy. The configuration is implemented in /etc/apache2/sites-available/default-ssl.conf.

Listing A.2: Reverse Proxy Configuration in Apache

```
1
2 <IfModule mod_ssl.c>
3      <VirtualHost_default_:10125>
4
5 ServerAdmin email@example.net
```

```
6  ServerName 192.168.43.125
7
8  LoadModule proxy_module modules/mod_proxy.so
9  LoadModule proxy_connect_module modules/mod_proxy_connect.so
10 LoadModule proxy_http_module modules/mod_proxy_http.so
11 LoadModule ssl_module modules/mod_ssl.so
12
13 DocumentRoot /var/www/html
14
15 <Proxy *>
16  Order deny,allow
17  Allow from all
18 </Proxy>
19
20 ErrorLog ${APACHE_LOG_DIR}/error.log
21 CustomLog ${APACHE_LOG_DIR}/access.log combined
22
23 SSLEngine on
24 SSLProxyEngine On
25 SSLProxyCheckPeerCN off
26 SSLProxyCheckPeerName off
27 SSLProxyCheckPeerExpire off
28 SSLCertificateFile /etc/apache2/ssl/certificates/apache.crt
29 SSLCertificateKeyFile /etc/apache2/ssl/certificates/apache.key
30
31 <FilesMatch "\.(cgi|shtml|phtml|php)$">
32 SSLOptions +StdEnvVars
33 </FilesMatch>
34 <Directory /usr/lib/cgi-bin>
35 SSLOptions +StdEnvVars
36 </Directory>
37
38 ProxyPass / https://192.168.43.125:10125/
39 ProxyPassReverse / https://192.168.43.125:10125/
40
41 </VirtualHost>
42 </IfModule>
43
```

```
44 # vim: syntax=apache ts=4 sw=4 sts=4 sr noet
```

Appendix B

Declarative Programming in JAPyL

B.1 JAPyL Declarative Security Examples

In JAPyL, the security mechanisms are typically declaratively specified in either the Jupyter Server or Apache as discussed in Section 2.1.2. Two examples demonstrated below include IP Whitelisting, Blacklisting and Security Headers implementation.

Listing B.1: Apache IP Whitelisting, Blacklisting

```
1
2 <Directory>
3
4     Order Allow,Deny
5     Allow from 192.168.43.15
6
7 </Directory>
```

Listing B.2: Apache Security Headers

```
1
2 <IfModule mod_headers.c>
3
4 strict-transport-security: max-age=31536000;
     includeSubDomains; preload
5 expect-ct: max-age=31536000; enforce
6 referrer-policy: no-referrer
7 referrer-policy: strict-origin-when-cross-origin
8 x-permitted-cross-domain-policies: none
9 x-xss-protection: 1; mode=block
10 x-content-type-options: nosniff
11 x-frame-options: allow
```

```
12
13 </IfModule>
```

B.2 Developer Effort - Links vs PHP

Although tierless languages generally simplify the development of web applications, there are some cases that are less than elegant, as discussed in Section 2.4.3. One example is database update operations in Links where all references to a Table Handle include the fields in the table. This introduces a large amount of redundancy in the code as it is not often that all columns in a database are updated simultaneously.

Listing B.3: PHP Code Example

```
1
2 function markMilestoneComplete ($id) { mysql query("UPDATE
     milestone SET completed = 1 WHERE milestoneid = $id")};
```

Listing B.4: Links Code Example

```
1
2 fun markMilestoneComplete (milestoneid) {
3 update (var milestone <-- tb milestone)
4 where (milestone.milestoneid == milestoneid)
5 set (
6 milestoneid = milestone.milestoneid,
7 projectid = milestone.projectid,
8 creator = milestone.creator,
9 summary = milestone.summary,
10 priority = milestone.priority,
11 date start = milestone.date start,
12 date end = milestone.date end,
13 completed = 1
14 );
15 }
```

The one line SQL statement using PHP is easy to comprehend and makes it clear which fields are being updated, whereas the Links code requires 14 lines to maintain clarity, yet the columns being updated are far less obvious. Even if only updating a single column, Links requires code to be written relating to columns in the table definition, increasing developer effort.

Appendix C

Microservices in JPyL Web Stack

C.1 JPyL Individual Port Config/Reverse Proxy Microservice

In this version of JPyL, all Flask microservices are individual with their own specific functionalities. This example shows the individual Port Config/Reverse Proxy microservice.

Listing C.1: JPyL Port Redirect

```
1
2  global port
3  port=10125 #current port in use
4
5  def portToggle():
6      port=10138 #port redirect
7
8  @app.route('/', methods=['POST', 'GET'],
9  defaults={'path': ''})
10
11 @app.route('/<path:path>')
12 def catch_all(path):
13
14 def portToggle()
15 return redirect("https://192.168.43.100:10138")
```

C.2 Reliability Experiment Code Fragments

This section contains listings for key code fragments used in the reliability experiments in Sections 3.3.3 & 3.3.4.

Listing C.2: Sleep Function in Apache Reverse Proxy Module

```
1
```

```
2  static int proxy_detect(request_rec *r)
3  {
4      void *sconf = r->server->module_config;
5      proxy_server_conf *conf =
6          (proxy_server_conf *)
7          ap_get_module_config
8          (sconf, & proxy_module);
9          sleep(5);
10
11       r->proxyreq = PROXYREQ_PROXY;
12         r->uri = r->unparsed_uri;
13         r->filename = apr_pstrcat
14         (r->pool, "proxy:", r->uri, NULL);
15         r->handler = "proxy-server";
16
17         return OK;
18 }
```

Listing C.3: Sleep Function in Flask Reverse Proxy Module

```
1
2  def main():
3
4  try:
5      print ('Running flask webserver on port ' + str(port))
6      app.run(host='0.0.0.0', port=port)
7      time.sleep(5)
8  except:
9      killPort()
10     print ('Relaunching ' + _name_ + 'app')
11     time.sleep(8)
12     main()
```

C.3 JPyL Port Config/Reverse Proxy Microservices Chain

In this version of JPyL, there are different classes of microservices. This example shows the chained Port Config/Reverse Proxy microservice.

Listing C.4: JPyL Port Redirect

```
1
2  #proxy.py
```

```
3
4  from flask import Flask, jsonify
5  import urllib
6
7  app = Flask(__name__)
8
9  global port
10 port=10125
11
12 print ('Running Flask Web Server On Port ' +str(port))
13
14 toggle =
       urllib.request.urlopen('http://192.168.43.199:5005/toggleport/').read().d
15 port = int(toggle)
16
17 print('The Flask Server Is Now Running On ' +str(port))
18
19 if __name__ == '__main__':
20     app.run(host='192.168.43.199', port=port, debug=True)
21
22
23 #portconfig.py
24
25 import sys
26 from flask import Flask
27
28 app = Flask(__name__)
29
30 @app.route('/toggleport/', methods = ['GET', 'POST'])
31
32 def portToggle():
33
34     port = 10155
35     return (str(port))
36
37 if __name__ == '__main__':
38   app.run(host='192.168.43.199', port=5005, debug=True)
```

Appendix D

PRS & PWS Sensor Code

D.1 Light Sensor Code for PRS

The following listing shows the code for the light sensor in PRS.

Listing D.1: PRS Light Sensor Code

```
1
2  from protocols import i2c_pb2
3  from common import krpc as implementations
4  from common import config
5  from polling_sensor import PollingSensor
6  import lightconnect
7
8  _DEVICE_ADDR = 0x39
9
10 class LightSensor(PollingSensor):
11     value_type = float
12     sensor_type = PollingSensor.SENSOR_TYPES.LIGHT
13     i2c = LightConnect.light_variables()
14
15     def get_sensor_value(self):
16         transaction = i2c_pb2.I2cTransaction()
17
18         cmd = 0b10100000
19
20         # Read Channel 0
21         command = transaction.commands.add()
22         command.action = command.READ_WORD_DATA
23         command.addr   = _DEVICE_ADDR
24         command.cmd    = cmd | 0xC
```

```
25
26          # Read Channel 1
27          command = transaction.commands.add()
28          command.action = command.READ_WORD_DATA
29          command.addr   = _DEVICE_ADDR
30          command.cmd    = cmd | 0xE
31
32          controller = implementations.create_controller()
33          results = LightConnect.light_results(self)
34
35          if results != None and len(results.data_read) ==
               len(transaction.commands):
36              ch0 = results.data_read[0]
37              ch1 = results.data_read[1]
38              return self._data_to_lux(ch0, ch1)
39          else:
40              print "Problem with response from i2c service: %s" %
                   results
41              return None
42
43      def _power_up(self):
44          transaction = i2c_pb2.I2cTransaction()
45
46          command = transaction.commands.add()
47          command.action = command.WRITE_BYTE_DATA
48          command.addr   = _DEVICE_ADDR
49          command.cmd    = 0x00
50          command.data   = 0x03
51
52          controller = implementations.create_controller()
53          self.i2c.Transaction(controller, transaction, None)
54          print "power up requested"
55
56      def _data_to_lux(self, ch0, ch1):
57          # Using data for T/FN/CL package
58          lux = 0
59
60          if (ch0 == 0):
61              return lux
62
63          ratio =  1.0 * ch1/ch0
```

```
64
65          if ratio < 0.50:
66              lux = 0.0304 * ch0 - 0.062 * ch0 * (ratio ** 1.4)
67          elif ratio < 0.61:
68              lux = 0.0224 * ch0 - 0.031 * ch1
69          elif ratio < 0.80:
70              lux = 0.0128 * ch0 - 0.0153 * ch1
71          elif ratio < 1.30:
72              lux = 0.00146 * ch0 - 0.00112 * ch1
73          else: #ratio > 1.30
74              Lux = 0
75          return lux
76
77      def additional_setup(self):
78
79          LightConnect.light_config(self)
80          self._power_up()
81
82  def main():
83      config.init('sensor_light')
84
85      sensor = LightSensor()
86      sensor.run()
87
88  if (__name__ == "__main__"):
89      main()
```

D.2 Light Sensor Code for PWS

The following listing shows the code for the light sensor in PWS.

Listing D.2: PWS Light Sensor Code

```
1
2  class LightConnect:
3
4    @staticmethod
5    def light_variables():
6
7        i2c = None
8        i2c_address = 0
```

```
9         i2c_transaction_length = 0
10        return i2c, i2c_address, i2c_transaction_length
11
12    @staticmethod
13    def light_values(self):
14
15        self.i2c.writeto(self.i2c_address, b"\x00") # power off
16        self._power_up()
17        self.set_mode(self.OP_SINGLE_HRES2)  # highest accuracy
18
19    @staticmethod
20    def light_results(self):
21
22        results = self.i2c.readfrom(self.i2c_address,
              self.i2c_transaction_length)
23        return results
24
25    @staticmethod
26    def light_power(self):
27
28        self.i2c.writeto(self.i2c_address, b"\x01")  # power on
29        self.i2c.writeto(self.i2c_address, b"\x07")  # reset
30
31    @staticmethod
32    def light_mode(self, mode):
33
34        self.i2c.writeto(self.i2c_address, bytes([self.mode]))  # set
              measurement mode
```

D.3 Comparing PRS & PWS Temperature Sensor Code

Both PRS & PWS implementations share similar codebases like objects and classes for equivalent comparison. The example below shows the implementation for the Temperature Sensors.

Listing D.3: PRS Temperature Sensor Implementation

```
1
2 from protocols import i2c_pb2
3 from common import krpc as implementations
4 from common import config
5 from polling_sensor import PollingSensor
```

```
6  import tempconnect
7
8  _DEVICE_ADDR = 0x48
9  _SCALE_FACTOR = 0.0625
10
11 class TemperatureSensor(PollingSensor):
12     value_type = float
13     sensor_type = PollingSensor.SENSOR_TYPES.TEMPERATURE
14     i2c = TempConnect.temp_variables()
15
16     def get_sensor_value(self):
17         transaction = i2c_pb2.I2cTransaction()
18
19         command = transaction.commands.add()
20         command.action = command.READ_WORD_DATA
21         command.addr = _DEVICE_ADDR
22         command.cmd = 0x00
23
24         controller = implementations.create_controller()
25         results = TempConnect.temp_results(self)
26
27         if results != None and len(results.data_read) ==
              len(transaction.commands):
28             data = self._swap_word_bytes(results.data_read[0])
29             return (data >> 4) * 0.0625
30         else:
31             print "Problem with response from i2c service: %s"
                  % results
32             return None
33
34     def _swap_word_bytes(self, word):
35         lsb = (word & 0xFF00) >> 8
36         msb = word & 0xFF
37         return (msb << 8) | lsb
38
39     def additional_setup(self)
40
41        TempConnect.temp_config(self)
```

```
42
43 def main():
44     config.init('sensor_temperature')
45
46     sensor = TemperatureSensor()
47     sensor.run()
48
49 if (__name__ == "__main__"):
50     main()
```

Listing D.4: PWS Temperature Sensor Implementation

```
1
2  from machine import Pin, I2C
3  import tempconnect
4  import time
5
6  class TemperatureSensor():
7
8      _DEVICE_ADDR = 0x45 #memory space to activate SHT30 Sensor
9
10     i2c = TempConnect.temp_connect()
11
12     mem1 = bytearray(2)
13     mem2 = bytearray(6)
14
15     #memory spaces accessed by SHT30()
16     reg1 = 0x30
17     reg2 = 0xA2
18     reg3 = 0x2c
19     reg4 = 0x06
20
21     def get_sensor_value(self):
22
23         results = TempConnect.temp_variables(self)
24         addr = self._DEVICE_ADDR
25
26         if results != None:
27
28             buf = self.mem1
```

```
29              buf[0] = self.reg1
30              buf[1] = self.reg2
31              results.writeto(addr, buf)
32              time.sleep_ms(1000)
33
34              data = self.mem2
35              buf[0] = self.reg3
36              buf[1] = self.reg4
37              results.writeto(addr, buf)
38              data = results.readfrom(addr, 6)
39
40              data2 = self._swap_word_bytes(data)
41              return data2
42
43          else:
44
45              print ("Problem with response from i2c service:
                    %s" % results)
46              return None
47
48      def _swap_word_bytes(self, data):
49
50          temp_raw = (data[0] << 8) + (data[1])
51          humi_raw = (data[2] << 8) + (data[3])
52          temp = 175 * temp_raw / 65535 - 45 #Formula Taken From
                MicroPython SHT30() Driver
53          humi = 100 * humi_raw / 65535 #Formula Taken From
                MicroPython SHT30() Driver
54
55          #return "humi = {:.2f}".format(humi)
56
57          #return "temp = {:.2f} humi = {:.2f}".format(temp,
                humi)
58
59          return ("temp = {:.2f} humi = {:.2f}".format(temp,
                humi))
60
61  #Activate the line of code to test sensor individually
```

```
62
63 #while True:
64   #time.sleep(5)
65   #test = TemperatureSensor()
66   #print (test.get_sensor_value())
67   #time.sleep(5)
```

Bibliography

[1] Peter Achten. Clean for Haskell98 Programmers – A Quick Reference Guide –. July 13 2007.

[2] Eyhab Al-Masri. Enhancing the microservices architecture for the internet of things. In *2018 IEEE International Conference on Big Data (Big Data)*, pages 5119–5125. IEEE, 2018.

[3] Kalev Alpernas, Yotam MY Feldman, and Hila Peleg. The wonderful wizard of loc: paying attention to the man behind the curtain of lines-of-code metrics. In *Proceedings of the 2020 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, pages 146–156, 2020.

[4] Gabriel Antoniu, Hinde Lilia Bouziane, Mathieu Jan, Christian Pérez, and Thierry Priol. Combining data sharing with the master–worker paradigm in the common component architecture. *Cluster Computing*, 10(3):265–276, 2007.

[5] George Athanasopoulos, Aphrodite Tsalgatidou, and Michael Pantazoglou. Interoperability among heterogeneous services. In *2006 IEEE International Conference on Services Computing (SCC'06)*, pages 174–181. IEEE, 2006.

[6] Sahin Aydin and Mehmet Nafiz Aydin. Semantic and syntactic interoperability for agricultural open-data platforms in the context of iot using crop-specific trait ontologies. *Applied Sciences*, 10(13):4460, 2020.

[7] Emmanuel Baccelli, Joerg Doerr, Ons Jallouli, Shinji Kikuchi, Andreas Morgenstern, Francisco Acosta Padilla, Kaspar Schleiser, and Ian Thomas. Reprogramming low-end iot devices from the cloud. In *2018 3rd Cloudification of the Internet of Things (CIoT)*, pages 1–6. IEEE, 2018.

[8] Vincent Balat. Ocsigen: Typing web interaction with objective caml. In *Proceedings of the 2006 Workshop on ML*, pages 84–94, 2006.

[9] Sourav Banerjee, Chinmay Chakraborty, and Sudipta Paul. Programming paradigm and the internet of things. In *Handbook of IoT and Big Data*, pages 145–163. CRC Press, 2019.

[10] Erik Barendsen and Sjaak Smetsers. Uniqueness typing for functional languages with graph rewriting semantics. *Mathematical Structures in Computer Science*, 6(6):579–612, 1996.

[11] Edd Barrett, Carl Friedrich Bolz, Lukas Diekmann, and Laurence Tratt. Fine-grained language composition: A case study. *arXiv preprint arXiv:1503.08623*, 2015.

[12] Don Batory and Sean O'malley. The design and implementation of hierarchical software systems with reusable components. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 1(4):355–398, 1992.

[13] Alvine Boaye Belle, Ghizlane El-Boussaidi, Christian Desrosiers, and Hafedh Mili. The layered architecture revisited: Is it an optimization problem? In *SEKE*, pages 344–349, 2013.

[14] Joel Bjornson, Anton Tayanovskyy, and Adam Granicz. Composing reactive guis in f# using websharper. In *Symposium on Implementation and Application of Functional Languages*, pages 203–216. Springer, 2010.

[15] Gordon S Blair, Massimo Paolucci, Paul Grace, and Nikolaos Georgantas. Interoperability in complex distributed systems. In *International School on Formal Methods for the Design of Computer, Communication and Software Systems*, pages 1–26. Springer, 2011.

[16] Yérom-David Bromberg, Paul Grace, and Laurent Réveillere. Starlink: runtime interoperability between heterogeneous middleware protocols. In *2011 31st International Conference on Distributed Computing Systems*, pages 446–455. IEEE, 2011.

[17] Björn Butzin, Frank Golatowski, and Dirk Timmermann. Microservices approach for the internet of things. In *2016 IEEE 21st International Conference on Emerging Technologies and Factory Automation (ETFA)*, pages 1–6. IEEE, 2016.

[18] Jordi Cabot. Wordpress: A content management system to democratize publishing. *IEEE Software*, 35(3):89–92, 2018.

[19] Keyan Cao, Yefan Liu, Gongjie Meng, and Qimeng Sun. An overview on edge computing research. *IEEE access*, 8:85714–85728, 2020.

[20] Valeria Cardellini, Emiliano Casalicchio, Michele Colajanni, and Philip S Yu. The state of the art in locally distributed web-server systems. *ACM Computing Surveys (CSUR)*, 34(2):263–311, 2002.

[21] Stephen Cass. The top programming languages: Our latest rankings put python on top-again-[careers]. *IEEE Spectrum*, 57(8):22–22, 2020.

[22] John L Casti. On system complexity: Identification, measurement, and management. In *Complexity, language, and life: Mathematical approaches*, pages 146–173. Springer, 1986.

[23] Emmanuel Cecchet, Anupam Chanda, Sameh Elnikety, Julie Marguerite, and Willy Zwaenepoel. Performance comparison of middleware architectures for generating dynamic web content. In *ACM/IFIP/USENIX International Conference on Distributed Systems Platforms and Open Distributed Processing*, pages 242–261. Springer, 2003.

[24] Ioannis K Chaniotis, Kyriakos-Ioannis D Kyriakou, and Nikolaos D Tselikas. Is node. js a viable option for building modern web applications? a performance evaluation study. *Computing*, 97(10):1023–1044, 2015.

[25] Robert Chatley, Alastair Donaldson, and Alan Mycroft. The next 7000 programming languages. In *Computing and Software Science*, pages 250–282. Springer, 2019.

[26] Stephen Chong, Jed Liu, Andrew C Myers, Xin Qi, Krishnaprasad Vikram, Lantian Zheng, and Xin Zheng. Secure web applications via automatic partitioning. *ACM SIGOPS Operating Systems Review*, 41(6):31–44, 2007.

[27] Ezra Cooper, Sam Lindley, Philip Wadler, and Jeremy Yallop. Links: Web programming without tiers. In *International Symposium on Formal Methods for Components and Objects*, pages 266–296. Springer, 2006.

[28] Richard Cooper. BBC online outage on saturday 19th july 2014, 2014. `https://www.bbc.co.uk/blogs/internet/entries/a37b0470-47d4-3991-82bb-a7d5b8803771`.

[29] Flavin Cristian. Understanding fault-tolerant distributed systems. *Comm. ACM*, 34(2):56–78, 1991.

[30] L Davis, Rose F Gamble, and Jamie Payton. The impact of component architectures on interoperability. *Journal of Systems and Software*, 61(1):31–45, 2002.

[31] Brad Dayley. *Node.js, MongoDB, and AngularJS web development*. Addison-Wesley Professional, 2014.

[32] Christophe De Troyer, Jens Nicolay, and Wolfgang De Meuter. First-class reactive programs for cps. In *Proceedings of the 4th ACM SIGPLAN International Workshop on Reactive and Event-Based Languages and Systems*, pages 21–26, 2017.

[33] Christophe De Troyer, Jens Nicolay, and Wolfgang De Meuter. Building iot systems using distributed first-class reactive programming. In *2018 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, pages 185–192. IEEE, 2018.

[34] Ryan Dewsbury. *Google web toolkit applications*. Pearson Education, 2007.

[35] Danny Dubé. Bit: A very compact scheme system for embedded applications. 2000.

[36] F Brito e Abreu and Miguel Goulão. Coupling and cohesion as modularization drivers: Are we being over-persuaded? In *Proceedings Fifth European Conference on Software Maintenance and Reengineering*, pages 47–57. IEEE, 2001.

[37] Toby Ehrenkranz and Jun Li. On the state of ip spoofing defense. *ACM Transactions on Internet Technology (TOIT)*, 9(2):1–29, 2009.

[38] Jeff Epstein, Andrew P Black, and Simon Peyton-Jones. Towards haskell in the cloud. In *Proceedings of the 4th ACM symposium on Haskell*, pages 118–129, 2011.

[39] Sinan Eski and Feza Buzluca. An automatic extraction approach: Transition to microservices architecture from monolithic application. In *Proceedings of the 19th International Conference on Agile Software Development: Companion*, pages 1–6, 2018.

[40] Bjorn N Freeman-Benson. Kaleidoscope: mixing objects, constraints, and imperative programming. In *ACM SIGPLAN Notices*, volume 25, pages 77–88. ACM, 1990.

[41] Peter Friess. *Internet of Things-Global Technological and Societal Trends From Smart Environments and Spaces to Green ICT*. River Publishers, 2011.

[42] Michael Furr and Jeffrey S Foster. Checking type safety of foreign function calls. *ACM SIGPLAN Notices*, 40(6):62–72, 2005.

[43] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, et al. An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 3–18, 2019.

[44] Qing Gao, Yingfei Xiong, Yaqing Mi, Lu Zhang, Weikun Yang, Zhaoping Zhou, Bing Xie, and Hong Mei. Safe memory-leak fixing for c programs. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 459–470. IEEE, 2015.

[45] Silvia Esparrachiari Ghirotti, Tanya Reilly, and Ashleigh Rentz. Tracking and controlling microservice dependencies. *Communications of the ACM*, 61(11):98–104, 2018.

[46] Will Glozer. wrk http benchmarking tool on github, 2019. https://github.com/wg/wrk.

[47] Google. Hipster-shop microservices demo, 2021. https://github.com/GoogleCloudPlatform/microservices-demo.

[48] Ian Gorton and Anna Liu. Evaluating the performance of ejb components. *IEEE Internet Computing*, 7(3):18–23, 2003.

[49] John John Gough and K John Gough. *Compiling for the. Net Common Language Runtime*. Prentice Hall PTR, 2001.

[50] Matthias Grimmer, Roland Schatz, Chris Seaton, Thomas Würthinger, Mikel Luján, and Hanspeter Mössenböck. Cross-language interoperability in a multi-language runtime. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 40(2):8, 2018.

[51] Jayavardhana Gubbi, Rajkumar Buyya, Slaven Marusic, and Marimuthu Palaniswami. Internet of things (iot): A vision, architectural elements, and future directions. *Future generation computer systems*, 29(7):1645–1660, 2013.

[52] Dinkar Gupta and Mrudul Palvankar. Pitfalls and challenges faced during a microservices architecture implementation. Technical report, 2020.

[53] Munish Gupta. *Akka essentials*. Packt Publishing Ltd, 2012.

[54] Andrea L Guzman. What is human-machine communication, anyway. *Human-machine communication: Rethinking communication, technology, and ourselves*, pages 1–28, 2018.

[55] Andreas Haas, Andreas Rossberg, Derek L Schuff, Ben L Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. Bringing the web up to speed with webassembly. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 185–200, 2017.

[56] Cordelia Hall, Kevin Hammond, Will Partain, Simon L Peyton Jones, and Philip Wadler. The glasgow haskell compiler: a retrospective. In *Functional Programming, Glasgow 1992*, pages 62–71. Springer, 1993.

[57] Wilhelm Hasselbring and Guido Steinacker. Microservice architectures for scalability, agility and reliability in e-commerce. In *2017 IEEE International Conference on Software Architecture Workshops (ICSAW)*, pages 243–246. IEEE, 2017.

[58] Nelson E Hastings and Paul A McLean. Tcp/ip spoofing fundamentals. In *Conference Proceedings of the 1996 IEEE Fifteenth Annual International Phoenix Conference on Computers and Communications*, pages 218–224. IEEE, 1996.

[59] Kristian Hentschel, Dejice Jacob, Jeremy Singer, and Matthew Chalmers. Supersensors: Raspberry pi devices for smart campus infrastructure. In *2016 IEEE 4th International conference on future internet of things and cloud (FiCloud)*, pages 58–62. IEEE, 2016.

[60] Victor Heorhiadi, Shriram Rajagopalan, Hani Jamjoom, Michael K Reiter, and Vyas Sekar. Gremlin: Systematic resilience testing of microservices. In *2016 IEEE 36th International Conference on Distributed Computing Systems (ICDCS)*, pages 57–66. IEEE, 2016.

[61] Hsi-Chuan Ho, Chun-Che Haung, and Heng-Li Yang. Development of modular services. In *2009 International Conference on New Trends in Information and Service Science*, pages 1215–1220. IEEE, 2009.

[62] Peng Huang, Chuanxiong Guo, Lidong Zhou, Jacob R Lorch, Yingnong Dang, Murali Chintalapati, and Randolph Yao. Gray failure: The achilles' heel of cloud-scale systems. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems*, pages 150–155, 2017.

[63] John Hughes. Why functional programming matters. *The computer journal*, 32(2):98–107, 1989.

[64] IBM. Drupal vs wordpress, 2021. https://www.ibm.com/cloud/learn/drupal-versus-wordpress-comparison.

[65] Nancy Ide and James Pustejovsky. What does interoperability mean, anyway? toward an operational definition of interoperability for language technology. In *Proceedings of the Second International Conference on Global Interoperability for Language Resources. Hong Kong, China*, 2010.

[66] IBM Market Development Insights. Microservices in the enterprise, 2021: Real benefits, worth the challenges. Technical report, Technical report, IBM, 2021.

[67] Christopher Ireland, David Bowers, Michael Newton, and Kevin Waugh. A classification of object-relational impedance mismatch. In *2009 First International Confernce on Advances in Databases, Knowledge, and Data Applications*, pages 36–43. IEEE, 2009.

[68] Pooyan Jamshidi, Claus Pahl, Nabor C Mendonça, James Lewis, and Stefan Tilkov. Microservices: The journey so far and challenges ahead. *IEEE Software*, 35(3):24–35, 2018.

[69] Miika Kalske, Niko Mäkitalo, and Tommi Mikkonen. Challenges when moving from monolith to microservice architecture. In *International Conference on Web Engineering*, pages 32–47. Springer, 2017.

[70] Mary Beth Kery, Marissa Radensky, Mahima Arya, Bonnie E John, and Brad A Myers. The story in the notebook: Exploratory data science using a literate programming tool. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*, pages 1–11, 2018.

[71] Pavneet Singh Kochhar, Dinusha Wijedasa, and David Lo. A large scale study of multiple programming languages and code quality. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, volume 1, pages 563–573. IEEE, 2016.

[72] Ravi Kishore Kodali and Kopulwar Shishir Mahesh. A low cost implementation of mqtt using esp8266. In *2016 2nd International Conference on Contemporary Computing and Informatics (IC3I)*, pages 404–408. IEEE, 2016.

[73] J Krishnamurthy and M Maheswaran. Programming frameworks for internet of things. In *Internet of Things*, pages 79–102. Elsevier, 2016.

[74] George Lawton. Web 2.0 creates security challenges. *Computer*, 40(10):13–16, 2007.

[75] Byeongcheol Lee, Ben Wiedermann, Martin Hirzel, Robert Grimm, and Kathryn S McKinley. Jinn: synthesizing dynamic bug detectors for foreign language interfaces. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 36–49, 2010.

[76] Euijong Lee, Young-Duk Seo, Se-Ra Oh, and Young-Gab Kim. A survey on standards for interoperability and security in the internet of things. *IEEE Communications Surveys & Tutorials*, 23(2):1020–1047, 2021.

[77] Jae Moon Lee. Development of translation telephone system by using mqtt protocol. *International Journal of Computational Vision and Robotics*, 9(6):599–608, 2019.

[78] James Lee and Brent Ware. *Open Source Web Development with LAMP: Using Linux, Apache, MySQL, Perl, and PHP*. Addison-Wesley Professional, 2003.

[79] Jong Kook Lee, Seung Jae Jung, Soo Dong Kim, Woo Hyun Jang, and Dong Han Ham. Component identification method with coupling and cohesion. In *Proceedings Eighth Asia-Pacific Software Engineering Conference*, pages 79–86. IEEE, 2001.

[80] Rasmus Lerdorf, Kevin Tatroe, Bob Kaehms, and Ric McGredy. *Programming Php*. " O'Reilly Media, Inc.", 2002.

[81] James Lewis and Martin Fowler. Microservices: A definition of a new architectural term, 2015. `https://martinfowler.com/articles/microservices.html`.

[82] Wing Hang Li, David R White, and Jeremy Singer. Jvm-hosted languages: they talk the talk, but do they walk the walk? In *Proceedings of the 2013 International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools*, pages 101–112. ACM, 2013.

[83] Yadong Li, Danlan Li, Wenqiang Cui, and Rui Zhang. Research based on osi model. In *2011 IEEE 3rd International Conference on Communication Software and Networks*, pages 554–557. IEEE, 2011.

[84] Yanjing Li, Young Moon Kim, Evelyn Mintarno, Donald S Gardner, and Subhasish Mitra. Overcoming early-life failure and aging for robust systems. *IEEE Design & Test of Computers*, 26(6):28–39, 2009.

[85] John W Lloyd. Practical advtanages of declarative programming. In *GULP-PRODE (1)*, pages 18–30, 1994.

[86] Panos Louridas. Component stacks for enterprise applications. *IEEE software*, 33(2):93–98, 2016.

[87] Mart Lubbers, Pieter Koopman, Adrian Ramsingh, Jeremy Singer, and Phil Trinder. Tiered versus tierless iot stacks: comparing smart campus software architectures. In *Proceedings of the 10th International Conference on the Internet of Things*, pages 1–9, 2020.

[88] Mart Lubbers, Pieter Koopman, Adrian Ramsingh, Jeremy Singer, and Phil Trinder. Could tierless languages reduce iot development grief? *ACM Transactions on Internet of Things*, 2022.

[89] Shang-Pin Ma, Chen-Yuan Fan, Yen Chuang, Wen-Tin Lee, Shin-Jie Lee, and Nien-Lin Hsueh. Using service dependency graph to analyze and test microservices. In *2018 IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC)*, volume 2, pages 81–86. IEEE, 2018.

[90] Alan MacCormack, John Rusnak, and Carliss Y Baldwin. The impact of component modularity on design evolution: Evidence from the software industry. *Harvard Business School Technology & Operations Mgt. Unit Research Paper*, (08-038), 2007.

[91] Amir Makhshari and Ali Mesbah. Iot bugs and development challenges. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 460–472. IEEE, 2021.

[92] Simon Malkowski, Markus Hedwig, and Calton Pu. Experimental evaluation of n-tier systems: Observation and analysis of multi-bottlenecks. In *2009 IEEE International Symposium on Workload Characterization (IISWC)*, pages 118–127. IEEE, 2009.

[93] Paul D Manuel and Jarallah AlGhamdi. A data-centric design for n-tier architecture. *Information Sciences*, 150(3-4):195–206, 2003.

[94] Philip Mayer and Alexander Bauer. An empirical analysis of the utilization of multiple programming languages in open source projects. In *Proceedings of the 19th International Conference on Evaluation and Assessment in Software Engineering*, pages 1–10, 2015.

[95] Manuel Mazzara, Antonio Bucchiarone, Nicola Dragoni, and Victor Rivera. Size matters: Microservices research and applications. In *Microservices*, pages 29–42. Springer, 2020.

[96] Daniele Mazzei, Giacomo Baldi, Gabriele Montelisciani, and Gualtiero Fantoni. A full stack for quick prototyping of iot solutions. *Annals of Telecommunications*, 73(7):439–449, 2018.

[97] Yuan Meng, Shenglin Zhang, Yongqian Sun, Ruru Zhang, Zhilong Hu, Yiyin Zhang, Chenyang Jia, Zhaogang Wang, and Dan Pei. Localizing failure root causes in a microservice through causality inference. In *2020 IEEE/ACM 28th International Symposium on Quality of Service (IWQoS)*, pages 1–10. IEEE, 2020.

[98] Marco Miglierina and Damian A Tamburri. Towards omnia: A monitoring factory for quality-aware devops. In *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering Companion*, pages 145–150, 2017.

[99] Michael Milligan. Interactive hpc gateways with jupyter and jupyterhub. In *Proceedings of the Practice and Experience in Advanced Research Computing 2017 on Sustainability, Success and Impact*, page 63. ACM, 2017.

[100] Kamal Sheel Mishra and Anil Kumar Tripathi. Some issues, challenges and problems of distributed software system. *International Journal of Computer Science and Information Technologies*, 5(4):4922–4925, 2014.

[101] Andrew Montalenti. Kafkapocalypse: a postmortem on our service outage, 2015. `https://blog.parse.ly/kafkapocalypse/`.

[102] Rebeca C Motta, Káthia M de Oliveira, and Guilherme H Travassos. On challenges in engineering iot software systems. In *Proceedings of the XXXII Brazilian symposium on software engineering*, pages 42–51, 2018.

[103] Keshab Nath, Sourish Dhar, and Subhash Basishtha. Web 1.0 to web 3.0-evolution of the web and its various challenges. In *2014 International Conference on Reliability Optimization and Information Technology (ICROIT)*, pages 86–89. IEEE, 2014.

[104] Netcraft. April 2021 web server survey, 2021. https://news.netcraft.com/archives/category/web-server-survey/.

[105] Efstratios Nikolaidis, Sophie Chen, Harley Cudney, Raphael T Haftka, and Raluca Rosca. Comparison of probability and possibility for design against catastrophic failure under uncertainty. *J. Mech. Des.*, 126(3):386–394, 2004.

[106] Chris Northwood. *The Full Stack Developer: Your Essential Guide to the Everyday Skills Expected of a Modern Full Stack Web Developer*. Springer, 2018.

[107] A Jefferson Offutt, Mary Jean Harrold, and Priyadarshan Kolte. A software metric system for module coupling. *Journal of Systems and Software*, 20(3):295–308, 1993.

[108] Joshua Ofoeda, Richard Boateng, and John Effah. Application programming interface (api) research: A review of the past to inform the future. *International Journal of Enterprise Information Systems (IJEIS)*, 15(3):76–95, 2019.

[109] Abrehet M Omer and Alexander Schill. Automatic management of cyclic dependency among web services. In *2011 14th IEEE International Conference on Computational Science and Engineering*, pages 44–51. IEEE, 2011.

[110] Arjan Oortgiese, John van Groningen, Peter Achten, and Rinus Plasmeijer. A distributed dynamic architecture for task oriented programming. In *Proceedings of the 29th Symposium on the Implementation and Application of Functional Programming Languages*, pages 1–12, 2017.

[111] Savan K Patel, VR Rathod, and Jigna B Prajapati. Performance analysis of content management systems-joomla, drupal and wordpress. *International Journal of Computer Applications*, 21(4):39–43, 2011.

[112] Piotr Pawlak, Bartosz Sakowicz, Piotr Mazur, and Andrzej Napieralski. Social network application based on google web toolkit. In *2009 10th International Conference-The Experience of Designing and Application of CAD Systems in Microelectronics*, pages 461–464. IEEE, 2009.

[113] Fernando Perez. Project jupyter, 2020. https://jupyter.org/.

[114] Laure Philips, Coen De Roover, Tom Van Cutsem, and Wolfgang De Meuter. Towards tierless web development without tierless languages. In *Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, pages 69–81, 2014.

[115] Paul Pichler, Barbara Weber, Stefan Zugal, Jakob Pinggera, Jan Mendling, and Hajo A Reijers. Imperative versus declarative process modeling languages: An empirical investigation. In *International conference on business process management*, pages 383–394. Springer, 2011.

[116] João Felipe Pimentel, Leonardo Murta, Vanessa Braganholo, and Juliana Freire. A large-scale study about quality and reproducibility of jupyter notebooks. In *2019 IEEE/ACM 16th international conference on mining software repositories (MSR)*, pages 507–517. IEEE, 2019.

[117] Sebastian Plamauer and Martin Langer. Evaluation of micropython as application layer programming language on cubesats. In *ARCS 2017; 30th International Conference on Architecture of Computing Systems*, pages 1–9. VDE, 2017.

[118] Maria Grazia Porcedda and David S Wall. Cascade and chain effects in big data cybercrime: Lessons from the talktalk hack. In *2019 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*, pages 443–452. IEEE, 2019.

[119] Alexander Power and Gerald Kotonya. A microservices architecture for reactive and proactive fault tolerance in iot systems. In *2018 IEEE 19th International Symposium on" A World of Wireless, Mobile and Multimedia Networks"(WoWMoM)*, pages 588–599. IEEE, 2018.

[120] PyPi. Chaos monkey engine project description, 2017. https://pypi.org/project/chaosmonkey/.

[121] Michael Rabinovich and Oliver Spatscheck. *Web caching and replication*, volume 67. Addison-Wesley Boston, USA, 2002.

[122] Taibur Rahman and Swarnendu Kumar Chakraborty. Provisioning technical interoperability within zigbee and ble in iot environment. In *2018 2nd International Conference on Electronics, Materials Engineering & Nano-Technology (IEMENTech)*, pages 1–4. IEEE, 2018.

[123] Adrian Ramsingh, Jeremy Singer, and Phil Trinder. Classifying the reliability of microservices architectures. In *Intl Conf on Web Information Systems and Technologies*, 2022.

[124] Adrian Ramsingh, Jeremy Singer, and Phil Trinder. Do fewer tiers mean fewer tears? eliminating web stack components to improve interoperability. *arXiv preprint arXiv:2207.08019*, 2022.

[125] Clean 3.0 Language Report. Uniqueness typing, 2020. [Online; accessed 02-February-2021].

[126] Ivan Ristic. *Apache security*. O'Reilly Media, 2005.

[127] Carlos Rodrigues, José Afonso, and Paulo Tomé. Mobile application webservice performance analysis: Restful services with json and xml. In *International Conference on ENTERprise Information Systems*, pages 162–169. Springer, 2011.

[128] Armin Ronacher. Flask: Web development one drop at a time, 2020. https://flask.palletsprojects.com/en/1.1.x/.

[129] Jarrett Rosenberg. Some misconceptions about lines of code. In *Proceedings fourth international software metrics symposium*, pages 137–142, Albuquerque, NM, USA, 1997. IEEE, IEEE.

[130] Davide Rossi. Consistency and availability in microservice architectures. In *Intl Conf on Web Information Systems and Technologies*, pages 39–55, 2018.

[131] Chaitanya K Rudrabhatla. Comparison of event choreography and orchestration techniques in microservice architecture. *International Journal of Advanced Computer Science and Applications*, 9(8):18–22, 2018.

[132] Adam Rule, Aurélien Tabard, and James D Hollan. Exploration and explanation in computational notebooks. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*, pages 1–12, 2018.

[133] Tasneem Salah, M Jamal Zemerly, Chan Yeob Yeun, Mahmoud Al-Qutayri, and Yousof Al-Hammadi. The evolution of distributed systems towards microservices architecture. In *2016 11th International Conference for Internet Technology and Secured Transactions (ICITST)*, pages 318–325. IEEE, 2016.

[134] Fred B Schneider. Synchronization in distributed programs. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 4(2):125–148, 1982.

[135] Manuel Serrano, Erick Gallesio, and Florian Loitsch. Hop: a language for programming the web 2. 0. In *OOPSLA Companion*, pages 975–985, 2006.

[136] Steven D Sheetz, David Henderson, and Linda Wallace. Understanding developer and manager perceptions of function points and source lines of code. *Journal of Systems and Software*, 82(9):1540–1549, 2009.

[137] Martin Shepperd. A critique of cyclomatic complexity as a software metric. *Software Engineering Journal*, 3(2):30–36, 1988.

[138] Kazuhiro Shibanai and Takuo Watanabe. Distributed functional reactive programming on actor-based runtime. In *Proceedings of the 8th ACM SIGPLAN International Workshop on Programming Based on Actors, Agents, and Decentralized Control*, pages 13–22, 2018.

[139] Hugo R Simpson. Layered architecture (s): Principles and practice in concurrent and distributed systems. In *Proceedings International Conference and Workshop on Engineering of Computer-Based Systems*, pages 312–320. IEEE, 1997.

[140] Jeremy Singer. Jvm versus clr: a comparative study. In *Proceedings of the 2nd international conference on Principles and practice of programming in Java*, pages 167–169. Citeseer, 2003.

[141] Jeremy Singer. Notes on notebooks: Is jupyter the bringer of jollity? In *Proceedings of the 2020 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, pages 180–186, 2020.

[142] Alessandro Sivieri, Luca Mottola, and Gianpaolo Cugola. Drop the phone and talk to the physical world: Programming the internet of things with erlang. In *2012 Third International Workshop on Software Engineering for Sensor Network Applications (SESENA)*, pages 8–14. IEEE, 2012.

[143] Jacopo Soldani, Damian Andrew Tamburri, and Willem-Jan Van Den Heuvel. The pains and gains of microservices: A systematic grey literature review. *Journal of Systems and Software*, 146:215–232, 2018.

[144] Isaac Strack. *Getting Started with Meteor. js JavaScript Framework*. Packt Publishing Ltd, 2015.

[145] Jurriën Stutterheim, Peter Achten, and Rinus Plasmeijer. Maintaining separation of concerns through task oriented software development. In *International Symposium on Trends in Functional Programming*, pages 19–38. Springer, 2017.

[146] Long Sun, Yan Li, and Raheel Ahmed Memon. An open iot framework based on microservices architecture. *China Communications*, 14(2):154–162, 2017.

[147] Toyotaro Suzumura, Scott Trent, Michiaki Tatsubori, Akihiko Tozawa, and Tamiya Onodera. Performance comparison of web service engines in php, java and c. In *2008 IEEE International Conference on Web Services*, pages 385–392. IEEE, 2008.

[148] Clemens Szyperski, Dominik Gruntz, and Stephan Murer. *Component software: beyond object-oriented programming*. Pearson Education, 2002.

[149] Davide Taibi and Valentina Lenarduzzi. On the definition of microservice bad smells. *IEEE software*, 35(3):56–62, 2018.

[150] Davide Taibi, Valentina Lenarduzzi, and Claus Pahl. Architectural patterns for microservices: A systematic mapping study. In *CLOSER*, pages 221–232, 2018.

[151] Giacomo Tanganelli, Carlo Vallati, and Enzo Mingozzi. Coapthon: Easy development of coap-based iot applications with python. In *2015 IEEE 2nd World Forum on Internet of Things (WF-IoT)*, pages 63–68. IEEE, 2015.

[152] CircuitPython Team. Circuitpython, 2022. [Online; accessed 2-March-2022].

[153] Giovanni Toffetti, Sandro Brunner, Martin Blöchlinger, Florian Dudouet, and Andrew Edmonds. An architecture for self-managing microservices. In *Proceedings of the 1st International Workshop on Automated Incident Management in Cloud*, pages 19–24, 2015.

[154] Nicholas H Tollervey. *Programming with MicroPython: embedded programming with microcontrollers and Python*. " O'Reilly Media, Inc.", 2017.

[155] Nachiappan Valliappan, Robert Krook, Alejandro Russo, and Koen Claessen. Towards secure iot programming in haskell. In *Proceedings of the 13th ACM SIGPLAN International Symposium on Haskell*, pages 136–150, 2020.

[156] Craig A VanLengen and John D Haney. Creating web services using asp. net. *Journal of Computing Sciences in Colleges*, 20(1):262–275, 2004.

[157] Vlad Vergu and Eelco Visser. Specializing a meta-interpreter: Jit compilation of dynsem specifications on the graal vm. In *Proceedings of the 15th International Conference on Managed Languages & Runtimes*, pages 1–14, 2018.

[158] Mario Villamizar, Oscar Garcés, Harold Castro, Mauricio Verano, Lorena Salamanca, Rubby Casallas, and Santiago Gil. Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud. In *2015 10th Computing Colombian Conference (10CCC)*, pages 583–590. IEEE, 2015.

[159] Jiafu Wan, Shenglong Tang, Zhaogang Shu, Di Li, Shiyong Wang, Muhammad Imran, and Athanasios V Vasilakos. Software-defined industrial internet of things in the context of industry 4.0. *IEEE Sensors Journal*, 16(20):7373–7380, 2016.

[160] Feng-Jian Wang and Faisal Fahmi. Constructing a service software with microservices. In *2018 IEEE World Congress on Services (SERVICES)*, pages 43–44. IEEE, 2018.

[161] Brent Ware et al. *Open source development with LAMP: using Linux, Apache, MySQL and PHP*. Addison-Wesley Longman Publishing Co., Inc., 2002.

[162] Micropython Official Website. Micropython differences from cpython, 2022. https://docs.micropython.org/en/latest/genrst/index.html.

[163] Peter Wegner. Interoperability. *ACM Computing Surveys (CSUR)*, 28(1):285–287, 1996.

[164] Pascal Weisenburger, Johannes Wirth, and Guido Salvaneschi. A survey of multitier programming. *ACM Computing Surveys (CSUR)*, 53(4):1–35, 2020.

[165] Eberhard Wolff. Why microservices fail: An experience report. Technical report, 2018.

[166] Andy Wu. Taking the cloud-native approach with microservices. *Ma-genic/Google Cloud Platform*, 2017.

[167] Tetsuya Yokotani and Yuya Sasaki. Comparison with http and mqtt on required network resources for iot. In *2016 international conference on control, electronics, renewable energy and communications (ICCEREC)*, pages 1–6. IEEE, 2016.

[168] Pamela Zave. A compositional approach to multiparadigm programming. *IEEE software*, 6(5):15–25, 1989.

[169] Steve Zdancewic, Lantian Zheng, Nathaniel Nystrom, and Andrew C Myers. Secure program partitioning. *ACM Transactions on Computer Systems (TOCS)*, 20(3):283–328, 2002.

[170] Xiang Zhou, Xin Peng, Tao Xie, Jun Sun, Chao Ji, Wenhai Li, and Dan Ding. Fault analysis and debugging of microservice systems: Industrial survey, benchmark system, and empirical study. *IEEE Transactions on Software Engineering*, 2018.

[171] Olaf Zimmermann. Microservices tenets. *Computer Science-Research and Development*, 32(3):301–310, 2017.

[172] Andrea Zonca and Robert S Sinkovits. Deploying jupyter notebooks at scale on xsede resources for science gateways and workshops. In *Proceedings of the Practice and Experience on Advanced Research Computing*, pages 1–7. 2018.

Glossary

abstraction also referred to as separation of concerns, this is where hierarchical composition of components in the stack abstracts the view of the system as a whole. Enough detail is provided to understand the roles of each layer and how the components relate to one another.. 98

architectural style defines the way of how to organise or structure the components of a system or platform.. 16

chained microservices a type of microservices implementation where some microservices are highly dependent or tightly coupled to others in order to function.. 91

Common Language Runtime (CLR) an intermediate concept between a compiler and an interpreter where a high level language is translated to butecode.. 90

Communication refers to software components in different physical machines being able to exchange messages or marshal data with each other over a network.. 17

component an independent, high performance, multi-threaded unit of software with a high level of abstraction, designed to perform a single function.. 1, 4, 5, 9, 13, 15–17, 60, 90–94, 99, 102

Coordination refers to software components being aware of their specific functionalities and working in sync with other components.. 17

CRS Clean, RaspberryPi Super Sensor. 59–61, 75, 77, 78, 80, 81, 83, 86–88, 105

CWS Clean, Wemos Super Sensor. 59–61, 75, 77, 78, 80, 83–88, 105

Defense-in-Depth a term in cybersecurity used to refer to using multiple layers of defense mechanism in order to protect a system.. 12

domain specific language (DSL) a programming language written to address a particular domain or support a specific set of tasks or concerns.. 29

dynamically typed refers to a programming language where the variable type is known only at run time.. 20

gray failures failures that are subtle at first but can eventually become catastrophic overtime depending on how long they are left undetected or unresolved.. 45

h2m human-to-computer interaction where data is generated based on human interacting with an application or system to perform a task. 15

heterogeneous interoperability describes all forms of interoperation in a multi-tier software stack as there are different components, languages and protocols integrated into a single platform.. 3, 4, 17

individual microservices a type of microservice that is loosely coupled and independently deployable.. 40

interoperability the ability of different components in an application to communicate, exchange information or share data in a unified format.. 16, 17, 100, 102

JAPyL webstack that comprises Jupyter, Apache, Python, Linux.. 3, 7, 15, 18, 33, 90, 91, 93, 96–102

JPL Jupyter, PHP, Linux Web Stack. 7, 92, 93, 96–103

JPyL Jupyter, Python, Linux Web Stack. 36, 92

LAMP Linux, Apache, MySQL, PHP. 1, 3, 9, 18, 22, 35

loose coupling components are detached or independent of one another.. 5

m2m machine-to-machine interaction where data is automatically generated based on machines or devices communicating with one another.. 15

MEAN MongoDB, Express.js, AngularJS, Node.js. 22, 90, 91

microservices a single application is developed as a suite of small services. Each one runs independently and communicates with lightweight mechanisms, usually via RESTful web services or messaging. The idea is to replace monolithic components in a stack with lightweight code libraries or services.. 5, 32

modularity allows a system to be structured as a set of components with clearly defined functionality. They can be implemented independently, and may be interchanged with other components that have similar functionality.. 91

multi-tier software stack organises interoperating components into tiers with specific functionalities and usually consists of a client-server architectural style.. 3–5, 17, 91, 101, 102

n-tier architecture that engineers web stacks into three layers: Presentation, Business Logic & Data Access.. 9, 92

Open Systems Interconnections (OSI) a logical and conceptual model that defines network communication as a layered server architecture system.. 15

polyglot programming the practice of writing code in multiple languages to capture additional functionality and efficiency not available in a single language... 4

programming languages can be seen as general purpose dynamic, scripting tools with a low level of abstraction necessary to deal with all the semantics, paradigms, libraries, modules, objects etc. needed in order to create the necessary interfaces to allow users to interact with the applications and components. 3, 19

PRS Python, RaspberryPi Super Sensor. 59–63, 66, 75–78, 80–82, 84–88, 91, 105

PWS (Micro)Python, Wemos Super Sensor. 3, 13, 15–18, 33, 59–63, 66, 75–78, 80–82, 84–88, 91, 105

semantic friction/impedance mismatch various technical challenges that can arise from integrating different languages and paradigms like type mismatches.. 4

single point of failure (SPOF) if one component fails, the entire application is interrupted even if the other components continue to operate.. 5

Single Responsbility Principle (SRP) a design principle that holds that each microservice should perform only one business function per service.. 34

statically typed refers to programming languages where the variable type is known before compile time.. 20

supersensor a device that has several different types of sensors attached to it for the capturing of several types of environmental data in real time.. 63

Task Oriented Programming (TOP) a declarative programming paradigm for constructing interactive distributed systems. Tasks are the basic blocks of TOP and represent work that needs to be done.. 67

tierless languages incorporate the use of a single language that synthesises all components/tiers in the software stack.. 5, 28, 91

web stack compilation of different software tools that work together to develop or support an application.. 1, 3