The Benefits of Tierless Elixir/Potato for Engineering IoT Systems

Solaris Li
Phil Trinder*
project.ocearia@gmail.com
Phil.Trinder@glasgow.ac.uk
University of Glasgow
Glasgow, United Kingdom

Christophe De Troyer
Vrije Universiteit Brussel
Brussels, Belgium
cdetroye@soft.vub.ac.be

Mart Lubbers
Radboud University
Nijmegen, Netherlands
mart@cs.ru.nl

Adrian Ramsingh
Sia Fusion Ltd
Glasgow, United Kingdom
adrian.ramsingh@siafusion.com

Abstract

IoT systems are increasingly pervasive, and developing, maintaining and ensuring the reliability of the software is challenging. IoT software is conventionally structured in multiple distributed tiers, where tiers use different programming languages and components that must interoperate. One way to minimise this complexity is to use a single *tierless* language to specify the entire IoT system. Tierless IoT languages require extremely sophisticated implementations, and are new and rare.

A previous study compared two Clean-based tierless implementations of a smart campus IoT system (CRS and CWS) with two conventional tiered Python implementations (PRS and PWS). It showed that tierless languages dramatically reduce development effort.

This paper describes a new implementation of the smart campus system in the Elixir/Potato tierless language (ERS), and compares ERS with the other implementations to show the following. (1) We provide further evidence that using a tierless IoT language reduces development effort. (2) We provide the first ever comparative study of two fundamentally different tierless IoT languages, i.e. we compare Elixir/Potato with Clean/iTask(mTask) using the ERS and CRS/CWS case studies. (3) We provide the first ever analysis of the software engineering costs of providing failure management in a tierless IoT language.

CCS Concepts: • **Software and its engineering → Distributed programming languages**.

Keywords: IoT, Elixir, Distributed System

ACM Reference Format:
Solaris Li, Phil Trinder, Christophe De Troyer, Mart Lubbers, and Adrian Ramsingh. 2024. The Benefits of Tierless Elixir/Potato for Engineering IoT Systems. In *Proceedings of the 23rd ACM SIGPLAN International Workshop on Erlang (Erlang '24), September 2, 2024, Milan, Italy*. ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3677995.3678197

1 Introduction

Conventional Internet of Things (IoT) software is complex and poses very significant software development and reliability challenges. In a typical IoT system a distributed set of sensors/actuators communicate with a server that in turn provides information/control to a distributed set of web clients. IoT software architectures typically compose multiple components organised in four or more tiers, with different programming languages and paradigms used in each tier [16]. A tiered architecture provides modularity as tiers can be loosely coupled, but there are significant challenges: (1) managing the semantic friction induced by simultaneously developing in multiple languages and paradigms; (2) correctly interoperating the languages & distributed components; (3) ensuring type safety across different tiers; and (4) handling a wide range of failures.

IoT sensor nodes may be microcontrollers like an ESP8266 based Wemos[1] with very limited compute resources, or supersensors: resource-rich single board computers like a Raspberry Pi [8]. Microcontrollers are the more demanding platform due to their limited resources like a tiny memory, executing on bare metal, etc.

BEAM languages naturally model such highly distributed systems, and Erlang/Elixir are widely used for engineering IoT systems [7, 10, 20]. One challenge for BEAM languages is the substantial memory footprint of the VM, meaning that they can only be deployed on supersensors. Projects like AtomVM and GRiSP address this by seeking to execute BEAM languages on microcontrollers[2]. Currently most Erlang/Elixir IoT implementations are tiered: they interoperate multiple components, often written in other languages, e.g. C device drivers, and HTML web content.

[1] https://www.wemos.cc/
[2] https://www.atomvm.net/ and https://www.grisp.org/

Tierless or multi-tier programming languages are a radical approach to reducing software development effort. In a tierless IoT language the entire IoT system is expressed as a single source in a single language. Hence an extremely sophisticated language implementation is required to generate the code for, distribute, and interoperate the components of each tier. There are mature tierless languages for some application domains, like Links [3], and Hop [19] for web applications. In contrast tierless IoT languages are more recent and less common. Combining the Potato library with Elixir provides *one of just three full-stack tierless IoT languages* (Section 2). Potato uses the Functional Reactive Programming (FRP) paradigm to react to and transmit IoT events [4].

A previous study has compared conventional tiered and tierless IoT technologies for both resource rich, and micro controller-based, sensor nodes [12, 13]. The basis for the comparison was the Glasgow University Smart Campus system (**GUSC**) outlined in Section 3.1. The tiered implementations primarily use Python and are denoted Python Raspberry Pi System (**PRS**), MicroPython Wemos System (**PWS**). The tierless implementations use Clean/iTask on the Raspberry Pi (**CRS**) and Clean/iTask/mTask on the Wemos (**CWS**). The study shows that the Clean tierless implementations require significantly lower development effort as they reduce code size, minimise interoperation, and maintain type safety.

Here we further investigate tierless IoT languages using on a new implementation of GUSC in Elixir/Potato (ERS) (Section 3). We analyse the differences between the Elixir/Potato, Python and Clean/iTask GUSC codebases. Here a key metric is Source Lines of Code (SLOC), a flawed but widely accepted proxy for software development and maintenance effort [17].

The paper makes three research contributions.

C1 We report a second study comparing tiered and tierless IoT software, further evidencing that tierless languages reduce development and maintenance effort (Section 4). **C1.a** Elixir/Potato requires far less development effort than Python: ERS uses 87% fewer (75 vs 562) lines of code, and less than 10% (3 vs 35) of the number of source files, than PRS. **C1.b** The tierless ERS developer experiences far less semantic friction than the PRS developer: they develop in fewer languages (3 rather than 6), and can more often use a declarative paradigm. **C1.c** The major differences in ERS and PRS code functionality are due to the idiomatic usage of libraries in ERS (3% vs 19% for database interfacing, 28% vs 10% for the web interface, and 32% vs 17% for communication). These results confirm the findings when comparing the tiered PWS and PRS codebases with the tierless CWS and CRS codebases [12].

C2 We provide the first ever comparative study of two fundamentally different Tierless IoT languages (Section 5). We compare Clean/iTask with Elixir/Potato using the ERS and CRS/CWS case studies. Tierless IoT languages are relatively new and rare, and the only previous study compares two closely related languages: Clean/iTask and Clean/iTask/mTask [12]. **C2.a** The codebases are very similar in size, with the Elixir/Potato developer writing just 16% less code (75 vs 89 SLOC). **C2.b** The ERS developer must overcome more semantic friction than the CRS/CWS developer. They must use three languages rather than one, and use a mix of imperative and declarative paradigms, where CRS/CWS are purely declarative. **C2.c** Using Functional Reactive Programming (FRP) in ERS makes the greatest difference in code functionality, requiring 6x more communication code than CRS (32% vs 4%) where communication is largely implicit. **C2.d** The Elixir/Potato and Clean/iTask implementations generate different distributed architectures for ERS and for CRS. For example ERS has just one database on the server, where CRS has a data store on the server with a lens on each sensor node. CRS has two implicit TCP communication channels between the sensor node and the server, where ERS has a single explicit UDP channel.

In summary, Clean/iTask(mTask) are radical tierless IoT languages, but are niche and lack a substantial software ecosystem. In contrast Elixir/Potato is more pragmatic: the developer must interoperate 3 languages, but benefits from a rich software ecosystem.

C3 We provide the first ever analysis of the software engineering costs of providing failure management in a tierless IoT language (Section 7). We explore four representative failures: sensor and sensor node hardware failures, sensor node software failure, and a UDP lost data communication failure. Three additional source files are required. The relative increase in code size to manage the failures is unsurprising at 71%, although the absolute amount of defensive code is fairly small at 53 SLOC, and much of it is boilerplate.

2 Related Work

2.1 Tiered IoT Software

Web applications are necessarily complex distributed systems, with client browsers interacting with a remote webserver and data store. Typical IoT applications are even more complex as they combine a web application with a second distributed system of sensor and actuator nodes that collect and aggregate data, operate on it, and communicate with the server [16]. To manage the complexity IoT software is traditionally organised into tiers or layers, each with clearly defined functionality. A common classification of IoT software tiers is illustrated in Figure 1, and comprises the following layers.

1. The perception layer: where data is collected, or the environment controlled, e.g. through sensors or actuators.
2. The network layer: responsible for communication between the sensor nodes and the server, often using protocols like MQTT.

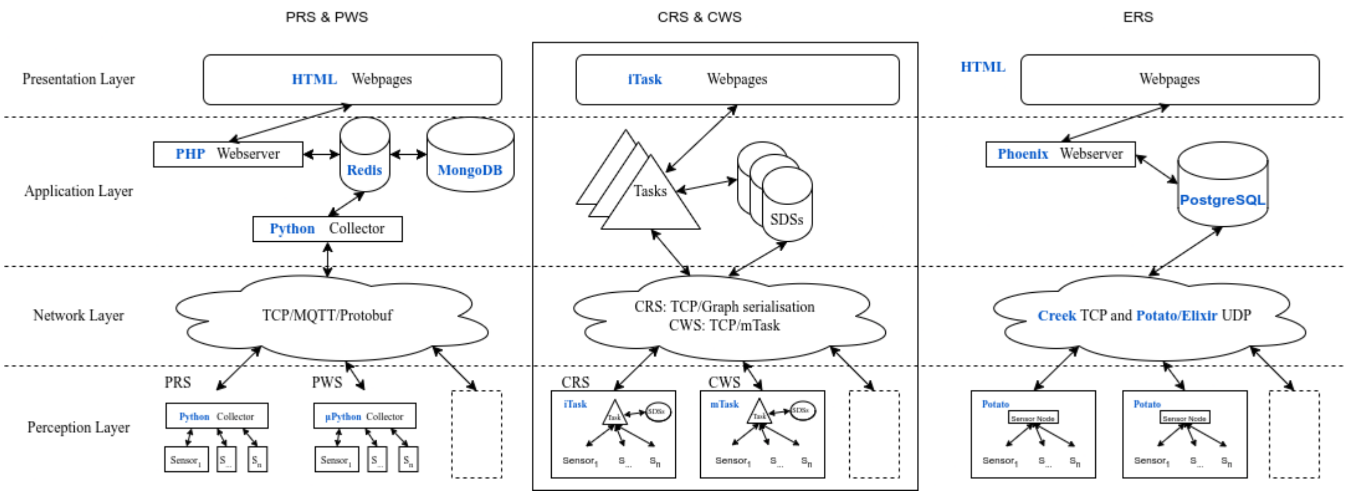


Figure 1. PRS, PWS, CRS, CWS, and ERS mapped to a four-tier IoT software architecture. Every box is the diagram denotes a source file or base. Bold blue text names the language or technology used in that source. The left-hand and middle diagrams taken from [12].

3. The application layer: acts as the interface between the presentation layer and the perception layer, storing and processing the data.
4. The presentation layer: utilises web components to interface between users and the IoT system.

Tiered architectures are adopted because they offer several benefits. A tiered architecture is modular, comprising set of components with clearly defined functionality that can be implemented independently, and may be interchanged. It provides abstraction over the functionality in each tier: how the perception tier works is not a concern for other tiers. It provides cohesion: all of the functionality related to a given task is contained in that tier, e.g. presentation functionality.

However a tiered IoT architecture poses very significant challenges for developers. The developer must interoperate components in multiple languages and paradigms, i.e. manage significant semantic friction. The developer must correctly interoperate the components, e.g. adhere to the API or communication protocols between components. To ensure correctness the developer must maintain type safety across a range of very different languages and diverse type systems. The developer must deal with the diverse failure modes of each component, and of component interoperations.

2.2 Tierless Languages

A radical approach to overcoming the challenges raised by tiered distributed software is to use a tierless, or multi-tier, programming language. Tierless languages minimise semantic friction by generating the code for all tiers, and all communication between tiers, from a single program. Typically, a tierless program uses a single language, paradigm, and type system, and the entire distributed system is simultaneously checked by the compiler.

Tierless languages have been developed for a range of distributed paradigms, including web applications, client–server applications, mobile applications, and generic distributed systems. A recent and substantial survey of these tierless technologies is available [24] and shows that there are established tierless languages for web development, for example Links [3], Clean/iTask [14] and Hop [19].

Tierless languages for IoT are both more recent and less common than for web applications. DSLs like Copilot [9] and Ivory [6] are embedded in a functional language and provide high-level programming for microcontrollers, e.g. guaranteeing strong typing and memory safety. While such DSLs simplify the perception layer, they do not cover the full IoT stack.

In an FRP paradigm the system reacts to discrete events: an excellent match for IoT systems that are inherently event-driven. Some DSLs, like Hailstorm [18] and Haski [23], use FRP to specify the perception layer. Elixir with the Potato library goes beyond other FRP languages to provide a full-stack tierless FRP IoT language for resource rich sensor nodes [4].

2.3 The Elixir/Potato Tierless IoT Language

Potato is an Elixir library that enables the tierless implementation of IoT software [4]. Elixir runs on the sophisticated BEAM VM that provides, *inter alia*, concurrency and fault tolerance. Potato is designed to reduce the accidental complexity inherent to the creation of IoT systems. Potato uses the (FRP) paradigm together with publish-subscribe communication, where every node can serve as a publisher and/or a subscriber. Communication is via the Creek imperative DSL that transmits data through streams, as we shall see in the ERS implementation (Section 3).

Key capabilities of Potato are methods to load new sensor nodes with basic functionality, and to send new programs for sensor nodes to execute, i.e. runtime deployment [22]. As Potato is built on simple high-level abstractions, there is minimal need to write boilerplate code. It also supports the full software lifecycle, reducing both developer and DevOps effort.

As Potato is a library for the established Elixir language, developers don't need to learn a new language. Moreover there is a rich ecosystem, with numerous libraries that support IoT functionalities. One example is the Circuits library for interfacing to sensors[3], and others are discussed below. Likewise the ecosystem is well provided with tools to support developers and DevOps. Elixir is better known, and has a larger user-base and ecosystem, than the Clean language we discuss next. However, Elixir does require the memory and compute-resource hungry BEAM VM, and hence an OS. These requirements prevent deployment on microcontrollers, restricting use to supersensors.

2.4 The Clean/iTask and Clean/iTask/mTask Tierless IoT Languages

Clean/iTask and Clean/iTask/mTask are currently the only other two full-stack tierless IoT languages. Clean is a statically typed functional programming language similar to Haskell: both languages are pure and non-strict [2]. Both iTask and mTask are DSLs embedded in Clean that adopt a Task-Oriented Programming (TOP) model for engineering interactive distributed systems [14]. The languages are tierless as from a single declarative description of tasks all of the required software components are generated. For example web servers, client code for browsers or IoT devices, and for their interoperation.

In Clean/iTask and Clean/mTask persistent data is maintained in Shared Data Sources (SDS) that may be distributed, and also provide parametric lenses or updatable views, on the data [5]. An SDS lens can notify tasks when data is updated. We see this in the Clean GUSC implementation where a server task is notified when sensor data changes in the `localSDS`, as shown in Figure 5(a).

The iTask DSL is intended for execution in a browser or on an operating system, and so can only be deployed on a supersensor. In contrast mTask is designed to be deployed on a bare metal microcontroller equipped with the mTask RTS/VM [11]. The GUSC codebases that we analyse in the remainder of the paper are implemented in Clean/iTask on a Raspberry Pi 3 (CRS) and in Clean/iTask/mTask on an ESP8266-based Wemos microcontroller (CWS).

[3]https://elixir-circuits.github.io/

2.5 Failure Handling in IoT

IoT systems typically comprise many small, low-power devices communicating over an unreliable network. So hardware, software, and network failures are all commonplace [1]. Hence, reliability is a key aspect of almost all IoT systems, to ensure that users can trust the system [15]. Implementing robust software failure handling is a key element of engineering a reliable IoT system, as it enables the handling of unexpected events and preventing unintended outcomes.

Failure management in tierless IoT software is discussed in general terms in [12]. However, we are not aware of any specific study of the engineering costs of providing failure management in tierless IoT software, such as we provide in Sections 6 and 7. This is likely because tierless IoT languages are so new.

3 ERS: A Smart Campus Case Study

3.1 Glasgow Smart Campus (GUSC) Case Study

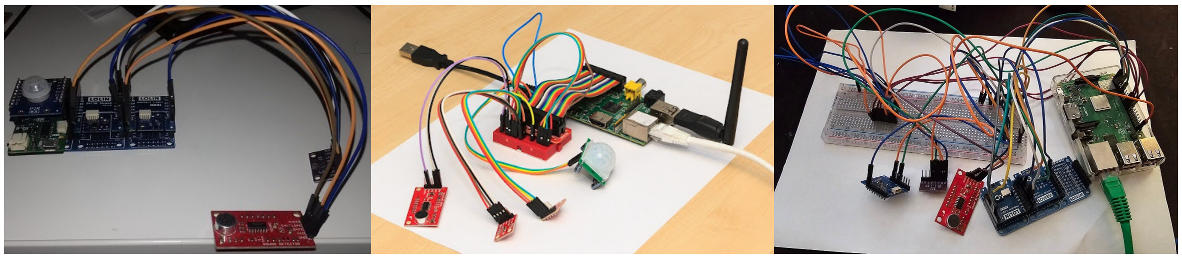
As part of a campus upgrade the University of Glasgow developed a prototype *smart campus* system to provide pervasive sensing infrastructure. The prototype uses modest commodity sensor nodes (i.e. Raspberry Pis) and low-cost, low-precision sensors for indoor environmental monitoring. The current set of sensors record temperature, humidity, sound, carbon dioxide, light and motion, and are shown in Figure 2. PRS sensor nodes have been deployed in 12 rooms in two buildings. PRS has an online data store, providing live access to sensor data through a RESTful API. This allows campus stakeholders to add functionality at a business layer above the layers that we consider here. To date, simple apps have been developed including room temperature monitors and campus utilization maps [8].

The following **high-level functional requirements** were specified by the GUSC project board. ERS meets these requirements, and is functionally equivalent to PRS, PWS, CRS and CWS.

1. be able to measure temperature and humidity as well as light intensity
2. scale to no more than 10 sensors per sensor node and investigate further sensor options like measuring sound levels
3. have access to communication channels like Wi-Fi, 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.

3.2 ERS Hardware

The main hardware components in ERS are the Raspberry Pi sensor node, the sensors, and the server (a computer). As a



(a) PWS & CWS **(b)** PRS & CRS **(c)** ERS

Figure 2. Exposed views of the GUSC sensor nodes: the Wemos on the left is used in PWS and CWS; the Raspberry Pi in the middle is used in PRS and CRS; the Raspberry Pi on the right is used in ERS

prototype the sensor node is connected to the sensors using a breadboard. The sensor node is connected to the server with an Ethernet cable. Figure 2 shows the hardware used in each GUSC implementation.

3.3 ERS Software

ERS uses the Potato FRP model, and the triggering events are when sensors send data to the sensor node. The sensor node then sends this data to the server through a stream. The server reacts to data it receives through the stream by uploading it to a database, which triggers a reaction on the website to update the values displayed. This chain reaction can be visualised by traversing from right to left in the deployment diagram in Figure 5(b).

The paragraphs below describe how each of the 4 IoT functions are implemented in ERS. Crucially these functions are integrated as part of a single Elixir/Potato source[4].

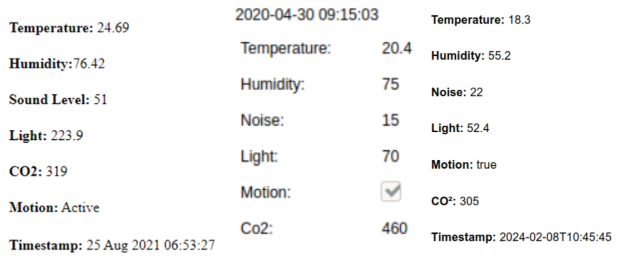
ERS Perception. The reading from each sensor is obtained in just a few lines of code using the Elixir Circuits hardware libraries[5].

ERS Network. When a sensor node connects to the server, the server sets up a data stream between itself and the sensor node as a directed acyclic graph (DAG) in the Creek library. In the DAG the source is the sensor node and the sink is the server [21], forming a stream. Once the sensor node has read data from all sensors, it sends the collated sensor data to the DAG, and the server receives it.

ERS Application. After receiving the sensor data, the server uploads the data to a "Measurements" table in an SQL database. This table has a field for each sensor reading and timestamps: temperature, humidity, noise, light, motion, CO2, time inserted, and time updated. The choice of an SQL DBMS is arbitrary: CRS uses SQL, where PRS uses the MongoDB time-series DBMS. Crucially for codebase comparison the Elixir interfaces to SQL and to MongoDB are very similar.

[4]ERS Source Code: https://zenodo.org/records/10952316
[5]https://elixir-circuits.github.io/

ERS Presentation. The ERS web interface uses the Phoenix LiveView library[6] that facilitates developing reactive websites. LiveView can display the value of a variable such that when the variable is updated the website also updates. Every five seconds the ERS webserver retrieves the latest record in the Measurements table and updates the associated variables, updating the values displayed on the website. Figure 3 shows the PRS, CRS and ERS websites.



Temperature: 24.69
Humidity:76.42
Sound Level: 51
Light: 223.9
CO2: 319
Motion: Active
Timestamp: 25 Aug 2021 06:53:27

2020-04-30 09:15:03

Temperature:	20.4
Humidity:	75
Noise:	15
Light:	70
Motion:	✓
Co2:	460

Temperature: 18.3
Humidity: 55.2
Noise: 22
Light: 52.4
Motion: true
CO²: 305
Timestamp: 2024-02-08T10:45:45

(a) PRS **(b)** CRS **(c)** ERS

Figure 3. The web interfaces provided by PRS, CRS, and ERS.

3.4 ERS Validation

Each of the sensors attached to the sensor node is confirmed to work. The value displayed on the website before and after changing the values of the sensor is compared. The sensor value is changed by performing actions such as shining a torch on the light sensor or clapping next to the sound sensor.

To enable fair comparison between the GUSC implementations, i.e. PRS, PWS, CRS, CWS and ERS, functional equivalence must be maintained. First, all implementations meet the GUSC functional requirements (Section 3.1). Second, each implementation uses five identical sensors to measure the same six values. Third, each website displays the same headings and data (Figure 3).

[6]https://hexdocs.pm/phoenix_live_view/Phoenix.LiveView.html

4 Comparing Tiered and Tierless Codebases

4.1 Memory and Power Consumption

Both maximum memory residency and power consumption are important operational characteristics for IoT sensor nodes. Here we discuss memory residency, and assume that there is negligible difference in the power consumption of the Raspberry Pis used by PRS, CRS and ERS, so between 1W and 2W depending on load.

Table 1. GUSC sensor node maximum memory residency (kibibytes).

PWS	PRS	CWS	CRS	ERS
20	3558	1	2726	20,698

Table 1 compares the maximum memory residency (in kibibytes) of the 5 GUSC implementations. ERS maximum memory residency is measured with the built-in ":erlang.memory()" Erlang function, and the ":etop.start()" Elixir function identifies the top memory-consuming processes.

The maximum memory residency of ERS (20698 KiB) is 5.8x greater than PRS (3558 KiB), which has the next greatest residency. PRS, CRS, and ERS are all designed to run on resource-rich sensor nodes so they all have large memory residencies (> 2.7MiB). In contrast, PWS and CWS are designed to minimise memory residency and have far smaller memory residencies <21KiB. Most of the ERS memory is occupied by the BEAM VM: 16221 KiB or 78%. Outside of the BEAM, the process using the most memory is the PostgreSQL database. This code is not needed on the sensor node, as the database resides on the server, however in this implementation all nodes must use the same codebase.

4.2 Comparing Tiered and Tierless IoT Implementations

This section compares tiered and tierless IoT software by comparing the tiered PWS and PRS with the tierless CWS, CRS and ERS. The ERS results are new, but the PWS, PRS[7], CWS and CRS[8] results are based on [12].

The original CWS and CRS interface to an SQL database to meet the GUSC requirements, and this interface makes up a significant portion of the codebase (78 SLOC, 47%). Idiomatic Clean data storage uses an SDS and lenses, requiring just 12 SLOC[9]. To make a fair comparison between idiomatic Clean/iTask(mTask) and Elixir/Potato we use Non SQL (NS) versions of the Clean implementations: CWS-NS and CRS-NS. A table including the analysis of original CWS and CRS codebases is available https://zenodo.org/records/11236782.

[7]PWS/PRS Source Code: https://zenodo.org/records/5081386

[8]CWS/CRS Source Code: https://zenodo.org/records/5040754

[9]CRS-NS and CWS-NS Source Code: https://zenodo.org/records/11142131

Code Size. is widely recognised as an approximate measure of the development and maintenance effort required for a software system [17]. Source Lines of Code (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 [17]. It is a simple measure, not dependent on some formula, and can be automatically computed.

Of course SLOC must be used carefully as it is easily influenced by programming style, language paradigm, and counting method. Here we are counting code to compare development effort, use the same idiomatic programming style in each component, and only count lines of code, omitting comments and blank lines.

4.2.1 Comparing Tiered and Tierless Codebase Sizes.

Code for each Functionality. Table 2 compares the tiered Python and tierless Clean/iTask and Elixir/Potato GUSC codebases using SLOC, and analysing the code by functionality. The last row reports the number of source files.

The functionalities are as follows. **Sensor Interface** code facilitates communication between the sensors and the sensor node software. **Sensor Node** code contains all other code on the sensor node that does not belong to any another category, such as control flow. **Manage Nodes** code coordinates sensor nodes, e.g. to add a new sensor node to the system. **Web Interface** code provides the web interface from the server, i.e. the presentation layer. **Database Interface** code communicates between the server and the database(s). **Communication** code transmits data and control between the server and the sensor nodes, and executes on both sensor node and server, i.e. the network layer.

ERS SLOC are measured by manually counting lines of code and verifying against the total SLOC and number of files reported by CLOC[10]. In PRS, PWS, CRS, and CWS, the SLOC is measured using tools such as pygount[11] [12].

Table 2 shows that with 75 SLOC, ERS has 87% less code than PRS, which has 576 SLOC. There is no functionality where ERS requires more code than PRS/PWS. These observations corroborate the findings for the Clean tierless languages in [12].

In each of the tierless implementations the code occupies just three source files, which is less than 10% of the files used by the two tiered implementations: 35 or 38 files. The difference arises as the tierless implementations mainly use a single language, so multiple functionalities are held in a file. In the tiered implementations, each functionality may be written in a different languages, requiring a separate file.

Proportional Code Coverage. Figure 4 compares the percentage of code occupied by each functionality in the tiered

[10]https://github.com/AlDanial/cloc

[11]https://pypi.org/project/pygount/

Table 2. Comparing Tiered and Tierless GUSC Code Sizes analysed by functionality. PWS, PRS, CWS-NS, and CRS-NS values based on Table 2 in [12]. Original SQL-based CWS and CRS converted to idiomatic SDS-based CWS-NS and CRS-NS. The full table including the original CWS and CRS can be found in https://zenodo.org/records/11236782.

		Tiered Python		Tierless Clean		Tierless Elixir
Code Location	Functionality	PWS	PRS	CWS-NS	CRS-NS	ERS
Sensor Node	Sensor Interface	52	57	11	11	11
	Sensor Node	178	183	9	4	3
Server	Manage Nodes	76		35	30	14
	Web Interface	56		28		21
	Database Interface	106		12		2
Communication	Communication	94	98	5	4	24
Total SLOC		562	576	100	89	75
No. Files		35	38	3	3	3

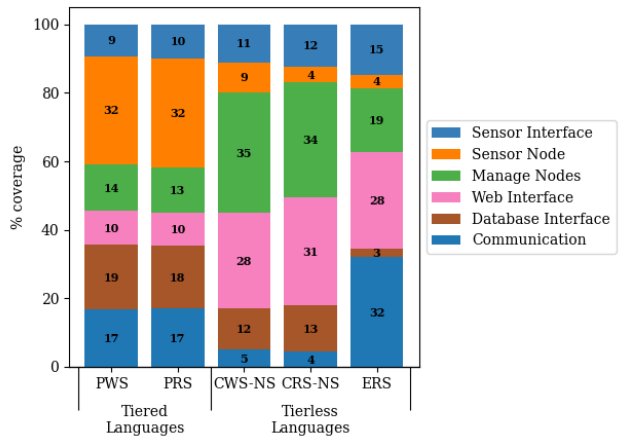


Figure 4. Comparing the percentage of code required to implement each functionality in tiered/tierless and resource-rich/constrained GUSC implementations. The full graph including the original CWS and CRS can be found in https://zenodo.org/records/11236782

Python and tierless Clean and Elixir GUSC codebases. In ERS, only 3% of the code is required for database interfacing, making it the smallest category. Minimal code is required as there is an Elixir library that expresses database queries as a single function without requiring SQL. In contrast the SQL interface in PRS occupies a larger proportion: 19%. In ERS, the web interface accounts for 28%, and communication for 32% of the SLOC. Although these proportions higher than in PRS/PWS (10% and 17%), the absolute amount of code is less (21 vs 56, and 24 vs 94 SLOC).

4.2.2 Comparing Tiered and Tierless Semantic Friction.

As a measure of the semantic friction that the developer manages Table 3 compares the languages and paradigms used in PWS, PRS, CWS-NS, CRS-NS, and ERS, again analysed by functionality. The bottom rows report the total languages and paradigms, and the number of imperative and declarative languages used. Here we use "language" to distinguish embedded DSLs from their host language: so Creek differs from Elixir; to distinguish dialects: so MicroPython differs from Python; and to distinguish frameworks/libraries: so Potato differs from Elixir.

ERS uses half the number of languages used in PRS (3 vs 6). Using multiple languages increases development effort as developers need to be fluent in more languages and paradigms, and additional interoperation is required.

Many argue that declarative languages are preferable to imperative languages, and both paradigms are used in ERS and PRS/PWS. However ERS is more declarative as 4 out of 6 functionalities are implemented in a declarative language, whereas in PRS/PWS only 2 of the 6 functionalities have some declarative code.

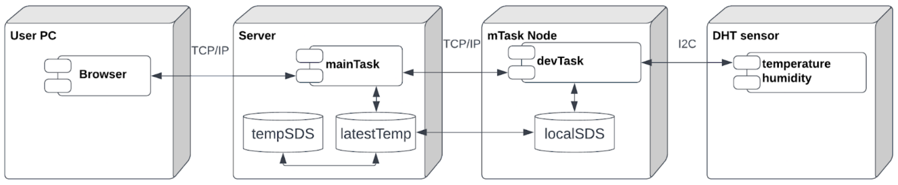
5 Comparing Tierless IoT Languages

Tierless IoT languages are very new and this section compares three of them using the ERS, CRS and CWS codebases. Crucially Elixir/Potato is fundamentally different from Clean/iTask and Clean/iTask/mTask. While both host languages are functional they follow different paradigms: Elixir is an impure distributed actor language, where Clean is a pure functional language. The libraries and DSLs also use different paradigms: Potato uses FRP while iTask and mTask use Task-Oriented Programming.

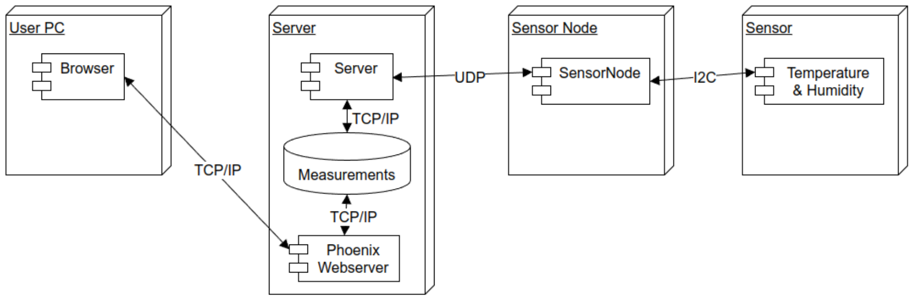
System Architecture. The different software architectures generated by Clean/iTask for CRTS and by Elixir/Potato for ERTS are shown in the deployment diagrams in Figure 5. CRTS/ERTS are simplified versions of CRS/ERS systems with a single temperature&humidity sensor on a single sensor node.

Table 3. Comparing the semantic friction by enumerating the languages and paradigms used in the GUSC implementations.

		Languages					Paradigms		
Code Location	Functionality	PWS	PRS	CWS-NS	CRS-NS	ERS	Python	Clean	Elixir
Sensor Node	Sensor Int.	μPython	Python	mTask	iTask	Creek	imp.	decl.	imp.
	Sensor Node	μPython	Python	mTask	iTask	Creek	imp.	decl.	imp.
Server	Manage Nodes	Python, JSON		iTask		Potato	imp.	decl.	decl.
	Web Int.	HTML, PHP		iTask		Phoenix	both	decl.	decl.
	Database Int.	Python,JSON,Redis		iTask		Potato	both	decl.	decl.
Communication	Communication	μPython	Python	iTask,mTask	iTask	Potato	imp.	decl.	decl.
	Total	7	6	2	1	3	2	1	2
	Imperative						6		2
	Declarative						2	6	4



(a) CRTS



(b) ERTS

Figure 5. Deployment diagrams for CRTS and ERTS, where CRTS/ERTS are CRS/ERS with only a temperature&humidity sensor on a single sensor node.

While necessarily similar as, for example, the code is deployed across the same three physical devices, there are some differences. Data is managed differently: ERTS has a single SQL database on the server storing all temperature readings. In contrast CRTS stores all temperature readings in the `tempSDS` on the server. The `latestTemp` lens is used to extract the latest temperature for display on the webpage. Moreover, each sensor node has a `localSDS` lens to access the data locally

The CRTS architecture induces additional implicit communication channels between the `localSDS` lens on each sensor node and the `tempSDS` on the server. In contrast the ERTS architecture reflects the single stream of data generated by the Potato FRP code. The stream flows from the sensor, to the sensor node, to the server, and to the website, i.e. leftwards through the diagram.

A second difference is that CRTS uses TCP/IP, whereas ERTS uses UDP, to transfer data between the server nodes and the server. UDP is used in the Potato library to broadcast messages to all IoT nodes, rather than using point-to-point TCP/IP connections. Using UDP means that data may be lost or arrive out of order. As the current GUSC implementations read the sensors every 5 seconds the effect of losing a set of readings is negligible. The impact of lost readings would be greater in a more realistic system that would read at longer intervals, say every 10 minutes.

Comparing Tierless Codebase Sizes. The last 3 columns of Table 2 compare the sizes of the ERS, CRS-NS and CWS-NS codebases. We primarily focus on CRS-NS as it, like ERS, is deployed on supersensors. They show that ERS has 16% less code than CRS-NS (75 vs 89 SLOC). A major difference between the two implementations can be seen in how managing the nodes in CRS-NS requires twice as much code as in ERS (30 vs 14 SLOC). In ERS the majority of node management is provided by the Potato library, and the developer only needs to create and broadcast a map that contains node identification information.

Another difference is in the database interface, where CRS-NS uses 12 and ERS uses just 2 SLOC. CRS-NS has an SDS and two lenses which, despite being concise, require more code than the Phoenix Ecto library used in ERS. Phoenix Ecto provides a single function call to read or write a record in the database.

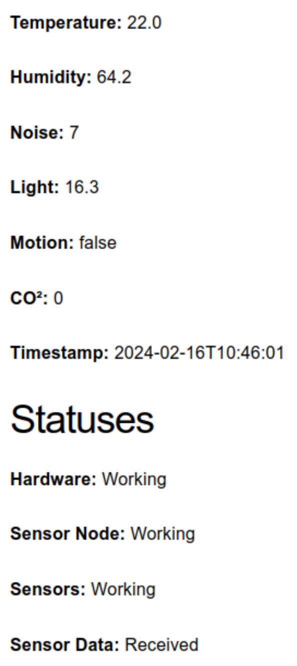
The three rightmost bars of Figure 4 compare the code proportions for each functionality in the CWS-NS, CRS-NS and ERS codebases. In ERS, the 32% of the code is used for communication, which is 6x more than CRS-NS (4%). This is primarily due to using the idiomatic but imperative Creek DSL for communication between the server node and the sensor nodes.

Comparing the Semantic Friction in the Tierless Implementations. ERS uses two more languages (3) than CRS-NS (1), as seen in the 6th and 7th columns of Table 3. The Creek communication DSL is embedded in Potato and used for streaming data from the sensor node to the server. It can be argued that the streams are familiar, given their similarity to streams in Java or Haskell. Phoenix LiveView was adopted for the website to exploit the capability for real-time updates, and as it turns database interactions into concise function calls. Fortunately both Creek and Phoenix are smoothly integrated with Elixir, minimising semantic friction.

The two rightmost columns of Table 3 show that where CRS-NS and CWS-NS only use declarative languages, the ERS developer must use both imperative and declarative. That is, they must use an imperative paradigm for 2 of the 6 functionalities. Needing to use both paradigms increases semantic friction for the ERS developer.

6 Engineering a Reliable Tierless IoT System

Reliability is a key aspect of most IoT systems, and most provide failure handling to manage unexpected events and prevent unintended outcomes. PRS, CRS, and ERS all provide some elementary failure management, e.g. if a sensor or sensor node fails the application layer is notified to report the failure.


Temperature: 22.0

Humidity: 64.2

Noise: 7

Light: 16.3

Motion: false

CO²: 0

Timestamp: 2024-02-16T10:46:01

Statuses

Hardware: Working

Sensor Node: Working

Sensors: Working

Sensor Data: Received

Figure 6. ERS-FH Website showing statuses of various systems when there are no failures.

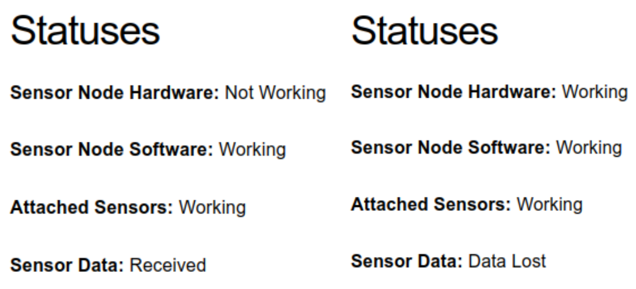
Here we consider adding failure management for three specific classes of IoT system failure to ERS. Specifically, we extend ERS to ERS with Failure Handling (ERS-FH[12]) that manages four representative failures. These are two hardware failures: a **sensor node hardware failure** arises if the Raspberry Pi sensor node disconnects from the server, while a **sensor failure** occurs when one of the five sensors attached to the sensor node fails. A **sensor node software failure** occurs when the Elixir/Potato program on the sensor node crashes. The network failure we consider is **lost sensor node data** where sensor node data is lost during the UDP transmission to the server.

Database and Website Extensions for Reliability. The ERS-FH database gains a new Statuses table to store the current status of each sensor node and its sensors. The records in this table have five fields: one for the sensor node ID and one for each potential failure. The ERS-FH website has a new section showing the statuses of each of the four potential failures, as retrieved from the Statuses table, and shown at the bottom of Figure 6. Figure 7 shows the statuses section of the website during two different failures - sensor node hardware, and lost sensor node data.

[12]ERS with Failure Handling (ERS-FH) Source Code: https://zenodo.org/records/10952316

Table 4. Comparing GUSC code sizes with failure handling (ERS-FH) and without (ERS).

Code Location	Functionality	ERS	ERS-FH
Sensor Node	Sensor Interface	11	11
	Sensor Node	3	3
Server	Manage Nodes	14	14
	Web Interface	21	32
	Database Interface	2	11
Communication	Communication	24	24
Failure Handling	Sensor Node Hardware		3
	Sensor Node Software		5
	Sensors (5 total)		20
	Lost Data		5
Total SLOC		75	128
No. Files		3	6


Statuses

Sensor Node Hardware: Not Working

Sensor Node Software: Working

Attached Sensors: Working

Sensor Data: Received

(a) Sensor Node Hardware

Statuses

Sensor Node Hardware: Working

Sensor Node Software: Working

Attached Sensors: Working

Sensor Data: Data Lost

(b) Lost Sensor Node Data

Figure 7. Screenshots of the ERS-FH website when (a) sensor node hardware has failed, and (b) when sensor node data is lost. Examples of the other two failures are available in https://zenodo.org/records/11236782.

Managing Failures in ERS-FH. When a sensor node connects or disconnects from the network, Potato calls a handler. To track the status of a node, a record is created in the Statuses table when a sensor node first connects to the network. When a connection is lost, this record is updated to show the failure status, which the website reacts to by changing the status displayed. This was validated by powering off the sensor node (Raspberry Pi), confirming that the website displays the correct (Not Working) status, reconnecting the Pi and checking the website again, as seen in Sub-figure (a) of Figure 7. Handling the other failures is similar, and is described in https://zenodo.org/records/11219193.

7 The Costs of Failure Management in a Tierless Language

7.1 SLOC per Functionality

Tierless IoT languages are very new, and this section provides the first analysis of failure handling in one by investigating the ERS-FH codebase that handles the four representative failures outlined above. Table 4 compares the Elixir/Potato codebase before (ERS) and after implementing failure handling (ERS-FH), again using SLOC and analysing the code by functionality. The last row reports the number of source files. Compared to Table 2 the table introduces a new functionality for each of the four failure types.

Implementing failure handling requires 71% extra code: the total SLOC for ERS-FH is 128 and for ERS is 75. The relative increase is typical of defensive code. However the absolute amount of defensive code is fairly small at 53 SLOC, and much of it is straightforward. Some failure handling simply pattern matches function results checking for success or failure. Likewise using powerful Elixir libraries in ERS-FH minimises the code size for failure handling, as it does in ERS.

The number of source files in the codebase increases by three to handle failures. Adding a small number of source files for failure handling is expected. The six source files in ERS-FH still remains far less than in the Python codebases, e.g. 35 in PWS. The functionalities of the addition source files are as follows. One file detects Potato nodes entering or leaving the network[13], and sensor node hardware and software failures. A second file detects and manages Potato node initialisation and restart. A third file uses Elixir supervision to monitor and restart the Elixir/Potato sensor node software if required.

[13]A Potato node is an Elixir node (VM) with a uninitialised Potato runtime.

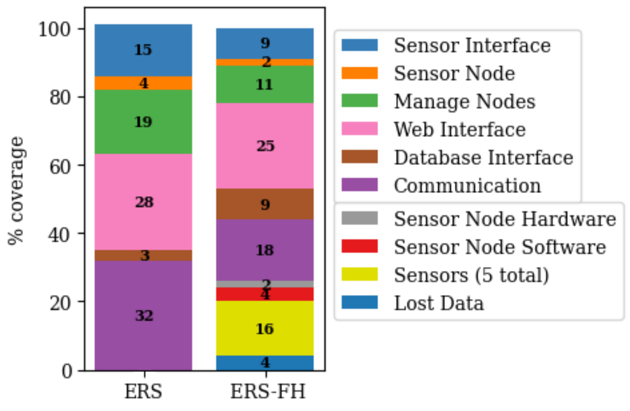


Figure 8. Comparing the percentage of code required to implement each functionality with failure handling (ERS-FH) and without (ERS).

7.2 Proportions of Failure Handling Code

Figure 8 compares the percentage of code required to implement each functionality in ERS and in ERS-FH. ERS-FH adds a functionality to ERS for each of the four failures handled. For failure handling, sensor failures require the most code (16%). This is because each sensor connection and data reading requires a pattern-match, typically adding four SLOC for each of the five sensors.

In ERS-FH the percentage of code for four of the six ERS functionalities decreases as no additional code is required. The percentage of Web Interface remains almost unchanged (28% vs 25%) reflecting the additional code to report failure statuses. The Database Interface increases by 3x in ERS-FH (from 3% to 9%) reflecting additional code to store failure statuses.

8 Conclusion

This study is based on ERS, a new tierless Elixir/Potato implementation of the Glasgow University Smart Campus (GUSC) system (Section 3). It provides **further evidence that developing IoT systems in a tierless language reduces development and maintenance effort** (Section 4). It does so by comparing the Elixir/Potato ERS with the Python PRS&PWS codebases to make the following three main findings. **(1)** Tierless Elixir/Potato requires far less development effort than tiered Python: ERS used 87% fewer (75 vs 562) Source Lines of Code (SLOC) in just 10% (3 vs 35) of the source files (Table 2). **(2)** The tierless ERS developer experiences far less semantic friction than the PRS developer: they develop in fewer languages (3 rather than 6), and can more often use a declarative paradigm: in 4 of 6 rather than 2 of 6 functionalities (Table 3). **(3)** The major differences in code for specific functionalities between ERS and PRS are due to the idiomatic usage of powerful libraries in ERS (3% vs 19% for database interfacing, 28% vs 10% for the web interface, and 32% vs 17% for communication) (Figure 4). These results confirm the results comparing tierless Clean CRS and CWS with tiered Python PRS and PWS [12].

Despite their significant benefits tierless IoT languages come with their own challenges. These include the need to learn new distributed programming abstractions like FRP or task-oriented programming, and if the developer wants to implement anything not included in the language they must use workarounds [12]. Moreover, both Potato and mTask are research languages, and community support is currently very limited. A specific drawback of Elixir/Potato is high memory residency compared to Python and Clean/iTask(/mTask). ERS occupies 5.8x more memory (21 MB) than the next largest: PRS (3.6 MB). So Elixir/Potato cannot be deployed on microcontrollers, and even resource-rich supersensors must be carefully selected. This may, however be addressed in future work.

The paper reports **the first ever comparative study of two fundamentally different Tierless IoT languages** using the ERS and CRS/CWS case studies (Section 5). Elixir/Pot-ato is very different from Clean/iTask and Clean/iTask/mTa-sk: Elixir is an impure distributed actor language, where Clean is purely functional. Moreover Potato uses FRP while iTask and mTask use Task-Oriented Programming.

The key differences are as follows. **(1)** The codebases are very similar in size, with the Elixir/Potato developer writing just 16% less code (75 vs 89 SLOC). The most significant differences are in how nodes are managed (14 vs 30 SLOC) and the database interfaces (2 vs 10 SLOC) (Table 2). **(2)** The ERS developer must overcome more semantic friction than the CRS/CWS developer. They must use more languages, i.e. three, rather than the single language used in CRS/CWS. They must also use a mix of imperative and declarative paradigms, while CRS/CWS are purely declarative (Table 3). **(3)** A major difference in code functionality is that ERS has 6x (32% vs 4%) more code for communication than CRS. This is due to using the imperative Creek FRP communication library (Figure 4). **(4)** Figure 5 illustrates the differences between the system architectures generated for ERS and CRS. Key differences are in how data is managed: ERS has a single database on the server where CRS/CWS has a server-side data store, with lenses on each sensor node; and in communication between the sensor nodes and the server (one vs two channels, and UDP vs TCP/IP).

Clean/iTask and Clean/iTask/mTask are radical tierless IoT languages based on a single purely functional language. They are however niche and lack a substantial software ecosystem. Elixir/Potato provides a less radical, and more pragmatic, approach to engineering tierless IoT software. While the developer must interoperate 3 languages, they do so with the support of a rich software ecosystem.

The study provides **the first ever analysis of the software engineering costs of providing failure management in a tierless IoT language** (Section 7). Managing four representative failures in ERS requires 71% additional code. While the relative increase in code size is unsurprising, the absolute amount of defensive code is fairly small at 53 SLOC (128 vs 75 in total) (Table 4), and much of it is straightforward (Figure 8). The number of source files increases from three to six, remaining far fewer than in the Python codebases (35 and 38 files).

Future Work. GUSC is an information harvesting system with one way data flow from the sensors via the sensor node and server to the web clients. Future work could investigate tierless IoT systems with *control*, so with information flowing from web clients to sensor nodes/actuators, e.g. allowing users to set a thermostat temperature from a website.

Further work could also explore an Erlang/Elixir tierless GUSC on a microcontroller, e.g. using GRiSP[14]. This would allow comparison with the PWS and CWS micro-controller-based tiered and tierless GUSC implementations [12].

Acknowledgments

Thanks to Kristian Hentschel, Dejice Jacob and Jeremy Singer who developed and maintain PRS, and to Pieter Koopman who collaborated on developing CWS and CRS. This work was funded in part by the UK EPSRC grant number EP/T014628.

References

[1] D. Anandayuvaraj and J. C. Davis. 2022. Reflecting on Recurring Failures in IoT Development. *Proc. IEEE/ACM International Conference on Automated Software Engineering* (2022), 1–5. 10.1145/3551349.3559545

[2] T. H. Brus, M. C. J. D. van Eekelen, M. O. van Leer, and M. J. Plasmeijer. 1987. Clean — A language for functional graph rewriting. In *Functional Programming Languages and Computer Architecture*, Gilles Kahn (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 364–384. https://doi.org/10.1007/3-540-18317-5_20

[3] Ezra Cooper, Sam Lindley, Philip Wadler, and Jeremy Yallop. 2007. Links: Web programming without tiers. *Formal Methods for Components and Objects, Springer, Berlin* (2007), 266–296. https://doi.org/10.1007/978-3-540-74792-5_12

[4] Christophe de Troyer, Jens Nicolay, and Wolfgang de Meuter. 2018. Building IoT Systems Using Distributed First-Class Reactive Programming. *2018 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)* (2018), 185–192. https://doi.org/10.1109/CloudCom2018.2018.00045

[5] László Domoszlai, Bas Lijnse, and Rinus Plasmeijer. 2014. Parametric Lenses: Change Notification for Bidirectional Lenses. In *Proc. Int. Symposium on Implementation and Application of Functional Languages (IFL '14)*. ACM, New York, NY, USA. https://doi.org/10.1145/2746325.2746333 event-place: Boston, MA, USA.

[6] Trevor Elliott *et al.* 2015. Guilt Free Ivory. In *Proc. ACM SIGPLAN Symposium on Haskell* (Vancouver, BC, Canada) *(Haskell '15)*. Association for Computing Machinery, New York, NY, USA, 189–200. https://doi.org/10.1145/2804302.2804318

[7] Geovane Fedrecheski, Laisa CP Costa, and Marcelo K Zuffo. 2016. Elixir programming language evaluation for IoT. In *2016 IEEE International Symposium on Consumer Electronics (ISCE)*. IEEE, 105–106.

[8] Kristian Hentschel, Dejice Jacob, Jeremy Singer, and Matthew Chalmers. 2016. Supersensors: Raspberry Pi devices for smart campus infrastructure. In *2016 IEEE 4th International Conference on Future Internet of Things and Cloud (FiCloud)*. IEEE, 58–62. https://doi.org/10.1109/FiCloud.2016.16

[9] Joey Hess. 2020. arduino-copilot: Arduino programming in Haskell using the Copilot stream DSL. //hackage.haskell.org/package/arduino-copilot

[10] Igor Kopestenski and Peter Van Roy. 2019. Erlang as an enabling technology for resilient general-purpose applications on edge IoT networks. In *Proc. ACM SIGPLAN Int. Workshop on Erlang*. 1–12.

[11] Mart Lubbers, Pieter Koopman, and Rinus Plasmeijer. 2019. Interpreting Task Oriented Programs on Tiny Computers. In *Implementation and Application of Functional Languages (IFL '19)*. ACM, Singapore, 1–12. https://doi.org/10.1145/3412932.3412936

[12] Mart Lubbers, Pieter Koopman, Adrian Ramsingh, Jeremy Singer, and Phil Trinder. 2023. Could Tierless Languages Reduce IoT Development Grief. *ACM Trans. Internet Things 4, 1 (February 2023)* (2023), 6:1–6:35. https://doi.org/10.1145/3572901

[13] Mart Lubbers, Pieter W. M. Koopman, Adrian Ramsingh, Jeremy Singer, and Phil Trinder. 2020. Tiered versus tierless IoT stacks: comparing smart campus software architectures. In *IoT '20, Malmö, Sweden, October 6-9, 2020*. ACM, 21:1–21:9. https://doi.org/10.1145/3410992.3411002

[14] Rinus Plasmeijer *et al.* 2012. Task-Oriented Programming in a Pure Functional Language. In *Proc. Symposium on Principles and Practice of Declarative Programming* (Leuven, Belgium) *(PPDP '12)*. Association for Computing Machinery, New York, NY, USA, 195–206. https://doi.org/10.1145/2370776.2370801

[15] D. Ratasich, F. Khalid, F. Geissler, R. Grosu, M. Shafique, and E. Bartocci. 2019. A Roadmap Toward the Resilient Internet of Things for Cyber-Physical Systems. *IEEE Access* 7 (2019), 13260–13283. 10.1109/ACCESS.2019.2891969

[16] Arvind Ravulavaru. 2018. *Enterprise internet of things handbook : build end-to-end IoT solutions using popular IoT platforms*. Packt Publishing, Birmingham, UK.

[17] J. Rosenberg. 1997. Some misconceptions about lines of code. In *International Software Metrics Symposium*. 137–142. https://doi.org/10.1109/METRIC.1997.637174

[18] Abhiroop Sarkar and Mary Sheeran. 2020. Hailstorm: A Statically-Typed, Purely Functional Language for IoT Applications. In *Proc. Principles and Practice of Declarative Programming* (Bologna, Italy) *(PPDP '20)*. ACM. https://doi.org/10.1145/3414080.3414092

[19] Manuel Serrano, Erick Gallesio, and Florian Loitsch. 2006. Hop: A language for programming the web 2.0. *Proc. Object-Oriented Programming Systems, Languages, and Applications (OOPSLA Companion'06)* ACM, Portland, Oregon, USA (2006), 975–985.

[20] Alessandro Sivieri, Luca Mottola, and Gianpaolo Cugola. 2012. Drop the phone and talk to the physical world: Programming the internet of things with Erlang. In *Workshop on Software Engineering for Sensor Network Applications (SESENA)*. IEEE, 8–14.

[21] Christophe De Troyer. 2022. *A meta-level architecture for stream-based programming languages and its applications in cyber-physical systems*. Ph. D. Dissertation. Vrije Universiteit Brussel, Belgium.

[22] Christophe De Troyer, Jens Nicolay, and Wolfgang De Meuter. 2017. First-class reactive programs for CPS. *Proc. ACM SIGPLAN International Workshop on Reactive and Event-Based Languages and Systems (October 2017)* (2017), 21–26.

[23] Nachiappan Valliappan *et al.* 2020. Towards Secure IoT Programming in Haskell. In *Proc. ACM SIGPLAN International Symposium on Haskell*. Association for Computing Machinery, New York, NY, USA, 136–150. https://doi.org/10.1145/3406088.3409027

[24] Pascal Weisenburger, Johannes Wirth, and Guido Salvaneschi. 2020. A Survey of Multitier Programming. *ACM Comput. Surv. 53, 4 (September 2020)* (2020), 81:1–81:35. https://doi.org/10.1145/3397495

[14] https://www.grisp.org/