

IMPERIAL

A PLATFORMISER FOR ETHICAL PLATFORMISATION

Author

MATTHEW S. SCOTT

CID: 01510365

Supervised by

PROF. JEREMY V. PITT

A Thesis submitted in fulfillment of requirements for the degree of
Doctor of Philosophy

Department of Electrical and Electronic Engineering
Imperial College London
2026

Abstract

The process of *platformisation* in the Digital Society has produced an ecosystem of large-scale, privately-owned, and monopolistic platforms. While this has produced many pro-social benefits, there are corresponding technical and anti-social drawbacks, such as limited opportunities for reuse, reduced support for domain-specific customisation, toxic platform dependency, algorithmic mis- and dis-information, and the entrenchment of asymmetric power relationships.

This thesis proposes a counter-process called *ethical platformisation* (EP). EP is intended to enable the ground-up self-organisation of social coordination by stakeholder communities, in a way that enables communities to incorporate ethical principles not just at the micro (platform) level, but at the macro (ecosystem) level too. In order to foster EP, this thesis also proposes a new and innovative tool for producing ethical platform ecosystems. We call this type of tool a *platformiser*. Accordingly, this thesis provides five contributions:

- We conceptualise EP as a counter-process to *platformisation* and provide eight **EP Principles** for how this process should be conducted;
- We conduct a critical analysis of the techno-politics of platformisation by investigating platform structuration across two dimensions of architecture and composition;
- We provide a proof of platform completeness by deriving an abstract platform formalism and corresponding mapping function for refinement into a concrete plugin architecture;
- We introduce *PlatformOcean* as one possible implementation of a platformiser by specifying the relevant algorithms and data structures for its constituent components; and
- We apply the *PlatformOcean* platformiser across six testcases, assessing the two dimensions of ecosystem scalability: platform cardinality and plugin cardinality.

This class of *platformiser* tool demonstrates that there is a *viable* alternative to the centralisation of power caused by the profit-driven leveraging of the digital platformisation process. Critically, the *PlatformOcean* implementation of a *platformiser* facilitates the counter-process of EP to produce ethically-inspired platform ecosystems for the purpose of community and stakeholder re-empowerment.

Declaration of Originality

I hereby declare that the work presented in this thesis is my own unless otherwise stated. To the best of my knowledge the work is original and ideas developed in collaboration with others have been appropriately referenced.

Copyright Declaration

The copyright of this thesis rests with the author. Unless otherwise indicated, its contents are licensed under a Creative Commons Attribution-NonCommercial 4.0 International Licence (CC BY-NC).

Under this licence, you may copy and redistribute the material in any medium or format. You may also create and distribute modified versions of the work. This is on the condition that: you credit the author and do not use it, or any derivative works, for a commercial purpose.

When reusing or sharing this work, ensure you make the licence terms clear to others by naming the licence and linking to the licence text. Where a work has been adapted, you should indicate that the work has been changed and describe those changes.

Please seek permission from the copyright holder for uses of this work that are not included in this licence or permitted under UK Copyright Law.

Contents

Abstract	i
Declaration of Originality	iii
Copyright Declaration	v
List of Acronyms	xi
1 Introduction	1
1.1 Motivation and Aims	1
1.2 Thesis Summary and Roadmap	4
1.3 Chapter Overviews	4
1.4 Statement of Contribution	7
2 Ethical Platformisation	11
2.1 Introduction	11
2.2 Platforms	12
2.3 Platformisation	14
2.4 Ethical Platformisation	16
2.4.1 Realisation of Positive Values	17
2.4.2 Reduction of Negative Values	18
2.4.3 Ethical Platformisation Principles	19
2.5 Platformisers	22
2.6 Summary	24
3 Platformiser Formalism	25
3.1 Introduction	25
3.2 Abstract Platformiser Formalism	27
3.3 Platformisation: Architecture and Composition	31
3.4 Architectural Layer	32
3.4.1 Centralised Client-Server	33
3.4.2 Distributed Client-Server	34

3.4.3	Blockchain	35
3.4.4	Blocklace	37
3.4.5	Summary – Trade-Off Framing	38
3.5	Compositional Layer	40
3.5.1	ActivityPub	41
3.5.2	Microkernel Paradigm	42
3.5.3	Summary – Trade-Off Framing	43
3.6	Convergence – <i>PlatformOcean</i>	44
3.6.1	Technical Architecture	45
3.6.2	Socio-Political Substrate	47
3.6.3	Parallel Trust Streams	48
3.7	Summary	49
4	Platform Completeness	53
4.1	Introduction	54
4.2	Background	55
4.2.1	Requirements Engineering for ‘Social Good’	56
4.2.2	Plugin Architectures	57
4.2.3	Constructive Mappings	58
4.3	Universal Platform Formalisation	60
4.3.1	Functional Requirements	60
4.3.2	Domain Data, Logic, and Interface Components	61
4.3.3	Interaction Contracts	62
4.3.4	Non-functional Requirements	63
4.3.5	Formal Platform Tuple	63
4.4	Plugin-Architecture Formalisation	64
4.4.1	Platform Kernel	65
4.4.2	Plugins	65
4.4.3	Interaction Contracts	66
4.4.4	Formal Plugin-Architecture Tuple	67
4.5	Constructive Proof of Universality	68
4.5.1	Plugins as Bounded Contexts	69
4.5.2	Kernel-Plugin Contracts	70
4.5.3	Kernel for Non-Functional Requirements	72

4.5.4	Constructive Mapping and Completeness Corollary	73
4.6	Examples	74
4.6.1	Non-Functional Requirements	74
4.6.2	Functional Requirements	75
4.6.3	Interaction Contracts	76
4.6.4	Methodology	77
4.7	Summary	78
5	Platformiser Implementation	81
5.1	Introduction	81
5.2	Meta-Platform	83
5.3	Platform	86
5.3.1	Backend	86
5.3.2	Frontend	90
5.4	Plugins	97
5.4.1	Security: Remote Code Execution	101
5.5	Software Development Kit	103
5.6	Software Stack	104
5.7	Comparison to Existing Extensible Web Architectures	106
5.7.1	Nextcloud	107
5.7.2	Drupal	108
5.7.3	WordPress	109
5.8	Summary	109
6	Platformiser Applications	113
6.1	Introduction	113
6.2	Case Study 1a: Twitter-like	115
6.2.1	Functional Requirements	116
6.2.2	Implementation	116
6.3	Case Study 1b: Pong-like	118
6.3.1	Functional Requirements	118
6.3.2	Implementation	119
6.4	Case Study 2: WhatsApp-like	121
6.4.1	Functional Requirements	122

6.4.2	Implementation	122
6.5	Case Study 3: Sporting Association Coordinator	122
6.5.1	Functional Requirements.	123
6.5.2	Implementation	123
6.6	Case Study 4a: Mind my Mind	126
6.6.1	Functional Requirements	126
6.6.2	Implementation	127
6.7	Case Study 4b: Climate-Smart School Meals Advisor	129
6.7.1	Functional Requirements	129
6.7.2	Implementation	130
6.8	Quantitative Profiling	133
6.9	Summary	136
7	Summary and Conclusions	139
7.1	Summary	139
7.2	Limitations	141
7.3	Further Work	143
7.4	Epilogue	145
A	Appendix: Supplementary Algorithms	147
	Bibliography	151

List of Acronyms

1

Introduction

Contents

1.1 Motivation and Aims	1
1.2 Thesis Summary and Roadmap	4
1.3 Chapter Overviews	4
1.4 Statement of Contribution	7

1.1 Motivation and Aims

Initial analysis of the *platform concept* used a network economics and business studies lens to identify a common medium for extracting externalities from both sides of a two-sided market [1]. The *platform* subsequently gained popularity in the business sphere, where commonalities were found and the abstract *platform* concept took concrete shape as a stable core that enabled complementary activity [2]. Following this, the *digital platform* emerged, manifesting the core/periphery divide from industrial platforms into a modular, reprogrammable software architecture [3], [4].

The process of *platformisation* characterises the increased proliferation of these platforms in various areas of the digital sphere. *Facebook*, *Twitter*, and *Instagram* now dominate the digital social media sphere; *Amazon*, *eBay*, and *Alibaba* now dominate retail and e-commerce marketplaces; and *Uber*, *AirBnB*, and *TaskRabbit* now dominate the so-called gig economy.

The process of platformisation has produced socio-technical ecosystems that have demonstrated both pro-social and anti-social consequences. Pro-socially, the increased uptake of digital plat-

forms has augmented the infrastructural dimension of software, thereby helping oppose pre-existing software oligopolies by providing *competitive* alternatives to governmental or quasi-governmental monopoly infrastructures [5].

Anti-socially, however, the centralisation of large-scale, monopolistic platforms at the application layer of the Internet has created opportunities incentivising platform builders to create lock-in for financial gain [5], either through *surveillance capitalism* [6], *addiction by design* [7], and frictionless design [8]. Some negative side-effects of platformisation, therefore, are not an accidental emergence but a *deliberate design choice*.

This thesis introduces *ethical platformisation* as a *counter-process*, i.e., a process that explicitly opposes and rejects the mainstream process, that can produce a malicious leveraging of platformisation. To realise positive values and reduce negative outcomes is a matter of *ethical choice* [9]. Therefore, in the same way that the *ISO 7000-2021* standard allows for the integration of ethics into system design, we argue that ethical principles *should* normatively, and *can* functionally, also be integrated into the design of the platform ecosystems generated by process of platformisation.

Ethically-inspired alternatives already exist to entrenched, monopolistic platform solutions. For example, *Signal* is a free, open-source, and highly-secure messaging app focused on user privacy. *Signal* deliberately avoids collecting user metadata, and is designed to serve as an alternative to *WhatsApp*. Per their mission statement, *Signal* is “an independent nonprofit [that is] not tied to any major tech companies, and [...] can never be acquired by one either” [10]. Furthermore *BlueSky* is a decentralised, open-source social media platform designed to serve as an alternative to *Twitter* by opposing their centralised, client-server-oriented business model. Moreover, users of *Bluesky* are able choose their own algorithms to avoid the manipulative, profit-driven engagement tactics fostered by *addiction-by-design* [7].

While these solutions serve as a convincing alternative to *individual* platform technologies, they fail to address the wider issues created by process of platformisation. Specifically, they are non-generic. This means that these solutions offer a static set of functions imposed by the platform designer, rather than dynamic functionality produced ground-up by the user community. Furthermore, the lack of genericity creates an issue of scope: addressing the malicious consequences of platformisation requires addressing any given platform in the entire ecosystem. Dynamic social psychologists have argued that bringing about behavioural change requires focussing on the social network and not individual minds [11]. Analogously, a single, bespoke solution is not sufficient; change must be enacted on a macro (ecosystem) level, rather than a micro (platform) level.

What is therefore required is a *platformiser* – i.e., a tool for conducting ethical platformisation. Rather than producing any standalone, ethical platform, the role of a platformiser is to produce and foster an entire ethical platform ecosystem. This thesis provides one such example of a platformiser in the form of *PlatformOcean*: an open-source tool for producing distributed, generic, plugin-based platform instances that can be customised bottom-up by the community for which they are designed. In helping supplant ‘unethical’ platformisation, however, this platformiser must be *viable* according to the following requirements:

- *arguable*: the case for a qualified form of platformisation should be coherent and consistent. Given the recent ISO standards, there is an argument to be made for *ethical* platformisation;
- *specifiable*: there should be a well-defined formal notation for defining the architectural components of a platform developed according to the principles of ethical platformisation;
- *computable*: it should be shown that the notation is complete, i.e., that just as a Turing Machine is a theoretical model of computation that serves as an abstraction of a physical computer, the notation should be a theoretical model that serves as an abstraction of any physical platform;
- *implementable*: although nobody builds Turing Machines (infinite tapes being in rather short supply), it is possible to map the logic of a Turing Machine to a physical device; therefore it should be possible to implement tools and toolchains that map the logic of the formal notation to an actual physical platform;
- *applicable*: it should be possible to build applications that satisfy particular requirements of a digital third place and its embedding in the ‘real-world’;
- *adoptable*: platforms should be adaptive and generative; that is, new functionality can be added as a community changes and learns, becoming more sophisticated; new platforms can be sourced directly from existing ones, lowering the barrier to entry; and unexpected new designs should be possible beyond the intention or imagination of the originators (i.e., generativity); and
- *acceptable*: the ISO definition of usability assesses the extent to which specified users can achieve defined goals with effectiveness, efficiency, recall and satisfaction in a certain context; similarly, users should be able to specify, operationalise and maintain their platforms within acceptable boundaries of energetic and financial investment.

1.2 Thesis Summary and Roadmap

This thesis constructs and validates *PlatformOcean*, one such example of a *platformiser*, explicitly. This verification is performed across five steps, spanning five chapters. *Arguability* is discussed in Chapter 2, following the introduction of the platform concept, *ethical platformisation* principles, and functional description of a *platformiser*. *Specifiability* is addressed in Chapter 3, by decomposing the *PlatformOcean platformiser* into its three constituent components. *Computability* is affirmed in Chapter 4 through a proof of platform completeness, mapping an abstract platform formalism onto a concrete plugin architecture. *Implementability* is confirmed in Chapter 5 through the explicit construction of the *PlatformOcean platformiser* and documentation of its relevant algorithms and data structures. *Applicability* is then demonstrated in Chapter 6 through a set of six *PlatformOcean* testcases, applying the platformiser across two dimensions of plugin and platform cardinality. We visualise this verification process as a roadmap in Figure 1.1.

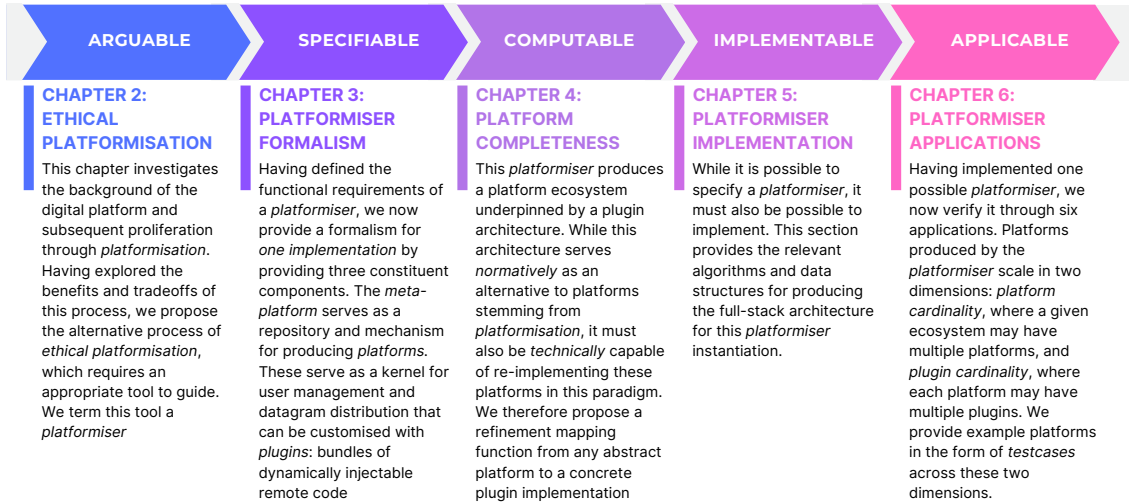


Figure 1.1: Thesis roadmap visualising steps required for establishing a viable alternative to mainstream platformisation

1.3 Chapter Overviews

Chapter 2 contextualises this research by first defining the *platform concept* that a platform ecosystem is built upon. This definition begins from the initial conception of a platform in the business sphere and subsequent evolution of the *digital platform* in the digital sphere. The process of *platformisation* is then defined in order to explain the increased penetration of these platforms in various arms of society. While there are pro-social benefits to the platformisation movement,

this chapter also investigates the *anti-social* effects of monopolisation and entrenchment of power asymmetries. In order to address these concerns, we define *ethical platformisation* as a counter-process to platformisation, and identify that it must be carried out with a set of eight **Ethical Platformisation (EP) Principles**. Following the definition of ethical platformisation and the specification of the prerequisite principles, we identify that an appropriate tool is required to guide the development of a platform ecosystem. We term this tool a *platformiser*.

Chapter 3 builds on the introduction of the amorphous *platformiser* tool from Chapter 2 by providing *PlatformOcean* – one concrete example of a *platformiser* – and giving it a formalism. To construct this formalism, we begin by providing five non-functional and three supra-functional requirements that a platformiser should perform. Beyond these eight design values, we also supply a further three technical and three political design values. These values are then coalesced into three, abstract components: a *platform*, a *meta-platform*, and a set of *modules*. By incorporating these three components, we argue that *PlatformOcean* functionally implements a *platformiser*. We consider the construction of the *PlatformOcean platformiser* across two dimensions: an architectural dimension, considering platform architectures from least to most centralised, and a compositional dimension, considering composability paradigms from top-down (as with the federation of static, pre-built sites) to bottom-up (using plugins to coalesce single pieces of functionality).

Chapter 4 uses the component formalism defined in Chapter 3 to provide the platforms generated by platformiser with a semantics. While a plugin-architecture exists *normatively* as a viable alternative to the privately owned and operated platforms stemming from process of platformisation, it must also be *technically* capable of re-implementing an existing platform in this paradigm. We use Evans’ *domain-driven design* to deconstruct a digital platform into a three-tuple, encapsulating its bounded contexts, interaction contracts, and non-functional requirements. We provide a similar decomposition for *PlatformOcean*’s plugin-based platform, deconstructing it into a two-tuple, encapsulating both its common, static kernel and dynamically-loadable plugins. A constructive mapping is then used to refine any abstract platform instance into an equivalent implementation as a *PlatformOcean* plugin-based platform. Having provided this mapping, we argue that the platforms produced from *PlatformOcean* are *platform complete*, by which we mean that there exists a *refinement mapping* at the architectural level.

Chapter 5 develops the *PlatformOcean meta-platform*, *platform*, and *module* components formalised in Chapter 3 into a concrete platformiser implementation available at <https://github.com/PlatformOcean>. Although we have demonstrated *semantically* in Chapter 4 that a plugin-based architecture is sufficiently expressive to be *platform complete*, a viable platformiser must

also be technically implementable. Therefore, for each of these components in the *PlatformOcean* platformiser, we specify their relevant algorithms and data structures. By explicitly constructing *PlatformOcean* – one example of a platformiser – we demonstrate that such a tool not only exists in principle, but also in practice. Therefore, it is possible to build a system to facilitate ethical platformisation.

Chapter 6 uses the platformiser built in Chapter 5 to produce six platformiser instantiation examples. *PlatformOcean* supports scalability in two dimensions: the cardinality of the number of platform instances, due to its implementation as a distributed client-server model; and the cardinality of the number of plugins for each instance, due to its implementation as a microkernel architecture. The *PlatformOcean* testcases therefore range from single-platform and single-plugin, to multi-platform and multi-plugin. This test suite is shown in Figure 1.2. For each of these test cases, we provide example data structures used for state synchronisation and screenshots for the platform’s user interface. Having tested *PlatformOcean* across these six test cases, we show not only that this platformiser is possible to build, but also possible to apply in the ‘real world’. These six testcases are:

- 1a. *Twitter-like* – A simple single-platform, single-plugin system where users can subscribe to a single ‘blog’ and send image- and text-based datagrams;
- 1b. *Pong-like* – A more advanced single-platform, single-plugin system requiring the dissemination of multiple, high-frequency datagrams to play *Pong*. This tests both the message-frequency tolerance of the system, and ability to abstractly represent plugin states;
2. *WhatsApp-like* – A multi-platform, single-plugin system analogous to *WhatsApp*: users can send and receive image- and text-based datagrams across different platform instances;
3. *Sporting-Association Coordinator* – A single-platform, multi-plugin system used for the coordination of a (legitimate) local sports team: users organise match fixtures, carpooling, laundry rota, and message exchange;
- 4a. *Mind My Mind* – A multi-platform, multi-plugin system used to coordinate (legitimate) students in a university. Plugins are designed based on the *Bartle taxonomy of player types* [12], allowing student groups to tailor their platform to fit their characteristics; and
- 4b. *Climate-Smart School Meals Advisor* – A more advanced multi-platform, multi-plugin system allowing for schools to coordinate the sourcing of local ingredients, and planning of sustainable school meals. Plugins allow for food-source database management, menu curation, and (AI-driven) menu recommendation. Platforms can be instantiated both in individual schools for meal planning, and multiple countries/constituencies for database management/availability.

Finally, Chapter 7 summarises the thesis and enumerates its limitations. We also provide opportunities for further work in the form of a project and research agenda, looking to evaluate *PlatformOcean* through a longitudinal, human-centric study, rather than short-term, software-centric studies. This chapter concludes with an epilogue, discussing how third-party users of *PlatformOcean* have built plugins and platforms; how *PlatformOcean* serves as the first step in a longer and broader research programme; and likening the modular approach of *PlatformOcean* to a separately-applicable design pattern.

1.4 Statement of Contribution

In total, this thesis offers five core contributions to knowledge:

1. **Conceptualisation of ethical platformisation** – we build on Poell’s definition of *platformisation* [13], supporting literature on its malicious consequences [6], [7], and ISO standards of system ethics [9] to introduce *ethical platformisation* (EP) as a counter process to platformisation. and eight corresponding **EP Principles**;
2. **Critical analysis of the techno-politics of platformisation** – we identify architecture and composition as two dimensions for the analysis of platform structure, identifying the benefits and drawbacks of these paradigms in supporting ethical platformisation;
3. **Proof of platform completeness** – we derive formalisms for both an abstract, amorphous platform and concrete plugin-based platform architecture. We define a mapping function to refine this abstract formalism into a corresponding implementation as a concrete, plugin-based platform. This demonstrates that a plugin architecture is platform complete, as it is sufficiently expressive to implement any conceivable platform;
4. **Implementation of a platformiser** – we introduce the concept of a platformiser as a tool to guide the process of ethical platformisation. We subsequently formalise and specify the relevant data structures and algorithms for a concrete implementation of one such platformiser called *PlatformOcean*; and
5. **Application of a platformiser** – we create six platform testcases for the *PlatformOcean* platformiser ranging from single-platform and single-plugin to multi-platform and multi-plugin.

From these contributions, we have therefore specified, formalised, and implemented one possible *platformiser* in *PlatformOcean*. Through this platformiser, we have demonstrated that a *viable* alternative exists to the centralisation of power caused by the malicious leveraging of the process of platformisation. Critically, this platformiser tool facilitates the counter-process of *ethical platformisation* to produce ethically-inspired platform ecosystems for the purpose of community and stakeholder re-empowerment.

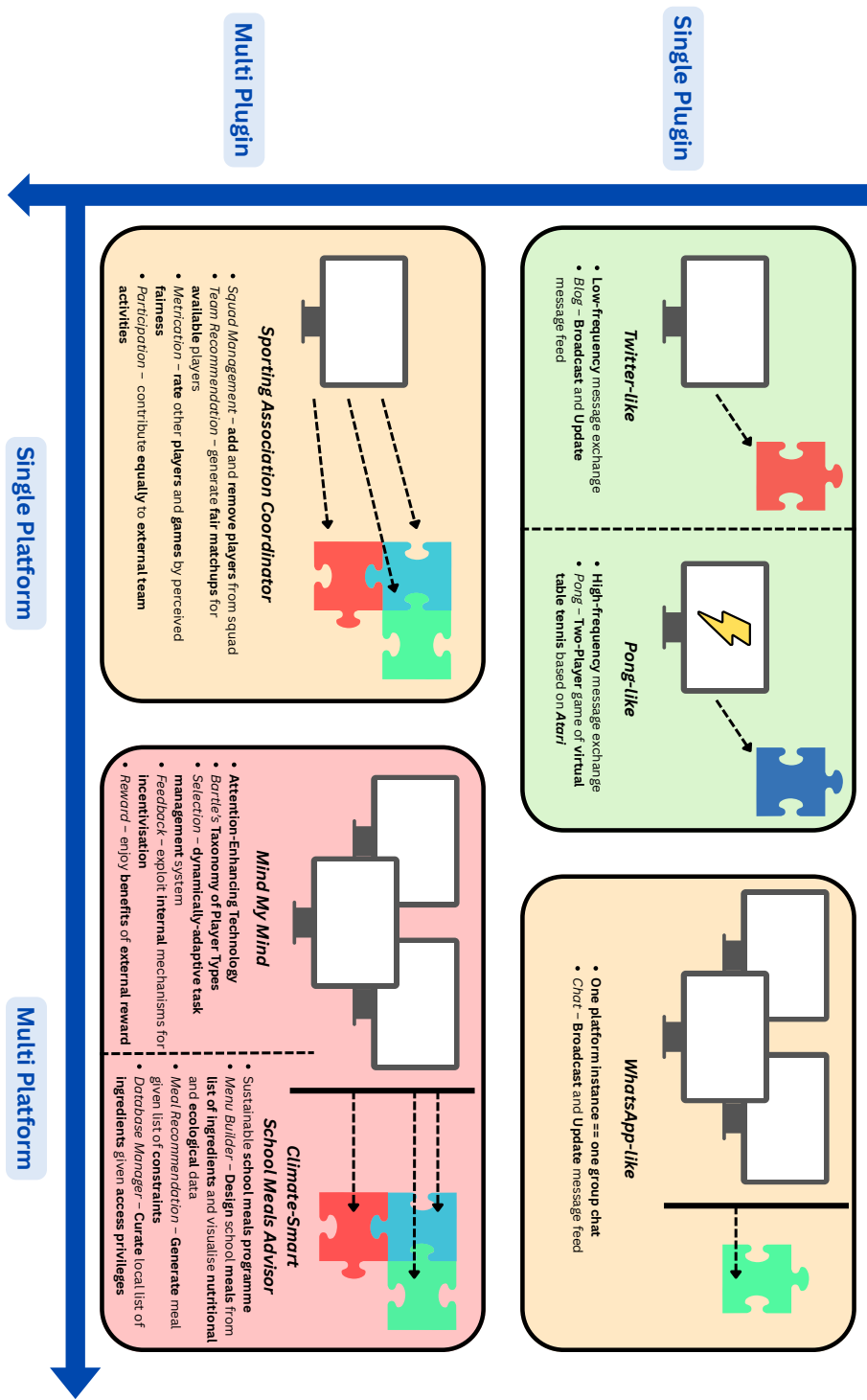


Figure 1.2: *PlatformOcean* application roadmap. Chapter 6 identifies two dimensions of scalability for evaluation: *plugin* cardinality, varying the number of plugins on a given platform; and *platform* cardinality, varying the number of platform instances in the ecosystem. A green background depicts the simplest platform implementation (i.e., single platform and single plugin). An orange background depicts a more challenging implementation (i.e., *either* multiple plugins or multiple platforms). A red background depicts the most challenging implementation (i.e., *both* multiple plugins and multiple platforms)

2

Ethical Platformisation

Contents

2.1 Introduction	11
2.2 Platforms	12
2.3 Platformisation	14
2.4 Ethical Platformisation	16
2.4.1 Realisation of Positive Values	17
2.4.2 Reduction of Negative Values	18
2.4.3 Ethical Platformisation Principles	19
2.5 Platformisers	22
2.6 Summary	24

2.1 Introduction

Early conceptualisations of the *platform* used a network economics and business studies lens to identify a common medium for extracting externalities from both sides of a two-sided market. The *platform* subsequently gained popularity in the business sphere, where commonalities were found and the *platform* took shape as a stable core that enabled complementary activity. Following this, the *digital platform* emerged, manifesting the core/periphery divide from industrial platforms into a modular, reprogrammable software architecture.

The rapid increase in popularity of the *digital platform* spawned the process of *platformisation*, whereby *digital platforms* have proliferated in various arms of the digital sphere. Despite the various positives resulting from this process, such as low-cost alternatives to governmental infras-

tructure, *platformisation* has also been linked to the creation of platforms that exhibit *surveillance capitalism*, *power asymmetries* between provider and stakeholder, and *addiction by design*.

Accordingly, this chapter stipulates the conditions for *ethical platformisation*, drawing inspiration from the *ISO 7000-2021* standards for ethical system design, grassroots technologies, and *Kantian ethics*. From these conditions, a set of eight *ethical platformisation principles* are abstracted and defined.

Having defined the necessary conditions for *ethical platformisation*, this chapter concludes by arguing that facilitating such a process requires an appropriate tool. This tool is defined as a *platformiser*, and conceptualised from existing examples in operating systems design, social media platforms, and video games.

2.2 Platforms

Rather than emerging as a concretely-defined technology, early conceptualisations of the *platform* served as coordination mechanisms used to foster *two-sided markets* [1], [13]. Two-sided markets are used to match two distinct groups in a financial relationship – conventionally *buyer* and *seller* – such that the value for one group increases proportionally to the number of participants in the other [3], [14]. Central to two-sided markets is the concept of a common substrate through which both sides of the market can interact, and from which externalities can be internalised [3], [14]. This substrate manifests in the form of a *platform*, with the principal function of ‘courting’ each side to make money overall [1]. This role is visualised in Figure 2.1. Ultimately, the ‘amorphous’ platform concept was used as a common medium, or coordination mechanism for unifying and mediating the activities of disaggregated groups in various areas of the business sphere [2], [3].

Commonalities were subsequently found in the manifestation of these platforms. In the industrial sphere, platforms were defined as comprising a stable, low-frequency *core*, and variable, high-frequency *periphery* [2]. This formulation was conducive for high-velocity innovation, due to the opportunities for distributed development – developers of the ‘core’ platform model could concentrate on making a minimally compatible substrate while sub-teams could focus on the modular periphery in parallel [2], [3]. With this lens, platforms underpin the structuration and governance of an entire *ecosystem* of interdependent actors [3].

Non-digital, industrial platforms presuppose strong modularisation while governed by an overarching design hierarchy [3], [16]. A direct analogy can be drawn to *digital* platforms, wherein

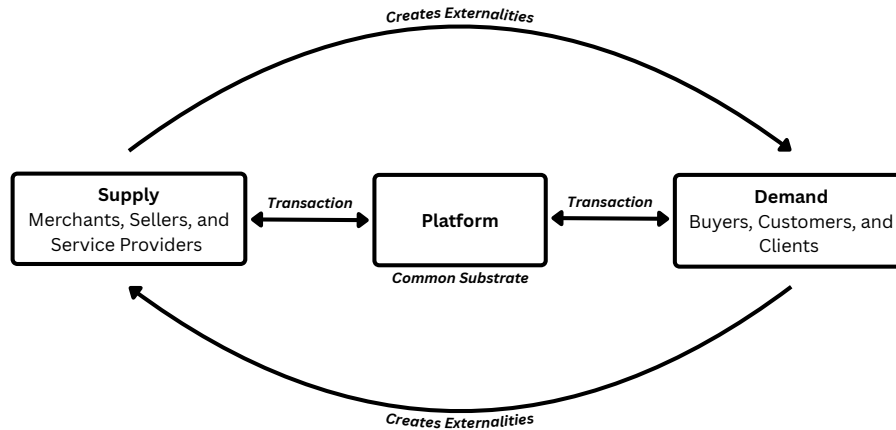


Figure 2.1: Platform serving as a *common substrate* when coordinating externalities in a two-sided market [3] (*cf.*, [15])

the necessary *editability* and *reprogrammability* of digital technologies [3], [17], [18], [19] can be achieved through a modular software architecture [4]. In this paradigm, a well-designed digital platform is analogous to a set of *Lego* bricks [4] – modular components of different shapes and sizes coalesce to form a single, cohesive, complex structure. With this formulation, users can readily separate and ‘plug’ modules from/into the platform (*cf.*, *hot-plugging* [20]), as well as independently version external modules without the need for concomitant updates to the core platform [4]. This approach to modularity directly maps to the decomposition of a platform into its *core* and *periphery* components [2].

The *Digital Platform Research Agenda* (DPRA [3]) defines the concept of a *digital platform* as the technical elements of software, hardware and associated organisational processes and standards. Beyond the digital platform is its (digital) *ecosystem*, defined by a collection of third-party complements to the core platform, including applications, modules, and micro-services. *Boundary resources*, including software tools and regulations, facilitate the arms’ length relationships between the involved parties such as service providers, application developers, and end users. *Platform openness* is then defined as the extent by which these boundary resources support the platform complements [3].

In *The Great Good Place*, Oldenberg decomposes significant locations in human life into three categories, or *places*. The *first place* defines the home, the *second place* defines the workplace, and the *third place* defines the locations that promote community building and, first and foremost, are responsible for uniting the neighbourhood [21]. Oldenburg supplies the example of the humble post office, to where members of the neighbourhood would commute every day and, despite the lack of seating, would converse (no matter how briefly) with one another [21]. Ultimately, third places are

intended to serve *everybody*. In this way, they become a catalyst for fostering a community where “everybody knows everybody”, serving as a kind of “mixer” where members of the community learn to be at ease with each other [21].

Despite the capacity for these *third places* to enrich social interactions, sense of community, and belonging, third places are disappearing [22], [23], due to increasing infrastructure costs and changing urban form. This disappearance was most notable in the arts, entertainment and recreation sector with a net change of -18.4% of third-place establishments in a seven-year span [23]. In light of this decline, digital platforms are increasingly becoming a kind of *digital third place*, with social media platforms such as *Discord* intentionally designed to foster these third-place experiences and serve as a viable alternative to its physical counterparts [24]. This transition to digital third places is also aided by increased support for neurodivergent stakeholders, say, as the presence of online avatars helps to alleviate social discomfort [25]. These digital third places therefore provide services at low-to-no cost, are permanently available, and are accessible from anywhere in the world. This rapid uptake of digital platforms in the social-media sphere and other aspects of the digital society is termed *platformisation*, and is discussed in the following section.

2.3 Platformisation

The *platformisation* movement denotes the process through which digital platforms have proliferated into the digital sphere [13]. This proliferation has been transformative, for better or worse, in both digital economic and business sectors, as well as more commonplace areas of the digital society [13].

On the one hand, the increased uptake of digital platforms in software engineering has created a “platformisation of infrastructures” and concomitant “infrastructuralisation of platforms” [5], [13]. By augmenting the infrastructural dimension of software in this way, platformisation has been credited with helping oppose pre-existing software oligopolies. Platforms stemming from the platformisation process became *competitive* alternatives to governmental or quasi-governmental monopoly infrastructures [5] due to their lower cost and increased dynamism [5], [13]. This facilitated the splintering of the modern infrastructural ideal following the emergence of neoliberal economies in the 1970s [5], [26]. Through a business-oriented lens, the value of monolithic platforms in particular has been integral in helping coalesce multiple otherwise-disparate software tools into a single, cohesive system [27]. This replaces the need for standalone, piecemeal functionality

with a unified architecture, thereby eliminating the need for domain-specific administration and streamlining the operationalisation process.

On the other hand, the commercialisation of digital platforms by process of platformisation comes with a cost. The trade-off of facilitating low-cost alternatives to entrenched infrastructures was a necessary transfer of wealth and responsibility instead to private enterprises [5]. Following this transfer, a second oligopoly has formed, where major platform companies have emerged as the platformised equivalents of the railroad, telephone, and electric utility monopolies [5], [13]. The monopolisation of large-scale software platforms in this way shifts the power to design critical features away from the stakeholder and towards the company owners. This creates an inherent tension between the pursuit of the ‘common good’ and the desire for profit [28]. In centralising this process, platforms often lack the civic engagement of the stakeholders themselves, causing software to be designed *for* users, and not *with* them [29].

The design and specification of these platforms therefore raises ethical concerns, whereby the inherently asymmetric provider-consumer power structure creates opportunities to disempower the stakeholder. The presence of AI systems in the platform infrastructure play an overarching role in the disinformation phenomenon prevalent in social media systems, among others [30]. These systems participate in *algorithmic misinformation* by facilitating the dissemination of disinformation to a targeted audience and at scale by malicious stakeholders [30], particularly when stakeholders are unable to opt out of this service.

Furthermore, this combination of power asymmetries, leveraging of AI systems, and opacity of proprietary code has enabled large-scale platforms to create a new economic model in which human stakeholders become the product. *Surveillance capitalism* [6], [31] defines the role of digital technologies in translating private experience into fungible commodities; companies maliciously leverage, or indeed *weaponise*, their digital platforms to extract and codify personal experience data as raw material. This data is subsequently used to predict future behaviours and sold in markets for targeted advertising, for example.

Similarly to how *surveillance capitalism* redefines platform stakeholders as products through commercialisation of their data, the *addiction by design* principle has been maliciously leveraged to produce the digital *attention economy*. In this economy, *attention* becomes the scarce resource, wherein technologies compete to capture and exploit human attention, rather than support stakeholder goals [32]. For digital platforms, profitability in this economy directly translates to maximising stakeholder engagement – platforms are incentivised to “as many people as possible to

spend as much time and attention as possible with one’s product or service” [32], thereby promoting addiction and lock-in. The participation of digital platforms in the *attention economy* strongly mirrors the *addiction by design* techniques employed in casinos, wherein the “mechanical rhythm” of electronic gambling machines puts players into a trance-like “machine zone” state [7]. Once in this state, gambling addicts play not to win but simply to keep playing, for as long as possible [7], compelled by game algorithms and machine ergonomics to maximise “time on device”. This is exemplified by *TikTok*, a social media platform specialising in short-form content, where pattern-finding and predictive-analysis algorithms circumvent the human decision-making pathways, delegating them to the platform instead [33], [34].

Ultimately, the consequences of platformisation have not been an accidental emergence, but a deliberate design choice. Platform builders are incentivised to create lock-in for financial gain [5], either through *surveillance capitalism* [6] or *addiction by design* [7]. Despite this, a key contention of this thesis is that *it does not have to be this way*. Accordingly, the following section envisions and defines a counter-process to platformisation entitled *ethical platformisation*.

2.4 Ethical Platformisation

The proliferation of software systems by process of platformisation has, in effect, generated a ‘system of systems’, whereby the modern technological landscape can be characterised by platforms not merely in isolation, by existing within an entire platform ecosystem. Accordingly, leveraging a *systems design* lens becomes a powerful tool to define *ethical platformisation*. In particular, the *ISO 7000-2021* standard provides a framework for addressing ethical concerns during system design. Specifically, this framework *integrates* these ethical considerations into the system design process, whereby *ethical* denotes supporting the realisation of positive values or the reduction of negative values [9]. This standard for ethical system design focuses on stakeholder engagement, human values, and the leveraging of ethical theory for value-sensitive, ethically-aligned design. Consequently, by involving a wide group of stakeholders in the design process, this standard helps organisations build more trustworthy systems [9].

While adopting a system design lens through the *ISO 7000-2021* standard provides a promising start for the definition of *ethical platformisation*, this approach lacks in scope. In particular, this standard considers platforms in isolation, rather than the curation of the entire ecosystem, or indeed the process by which the system is generated. Furthermore, while the *ISO 7000-2021*

standard establishes a set of *processes* for the inclusion of ethical values, it does not give specific guidance on the design of *algorithms* to apply such ethical values [9].

Accordingly, if ethical system design per the *ISO 7000-2021* standard [9] defines the process for incorporating human ethics into isolated systems, and platformisation defines the process by which platforms proliferate into the digital sphere [13], then *ethical platformisation* may be understood as the process by which human ethics are intentionally incorporated into the design, deployment, and governance of the entire platform ecosystem.

To incorporate human ethics into the platformisation process, we apply the definition of *ethics* from the *ISO 7000-2021* standard to produce two subsets of design principles for *ethical platformisation*. In Section 2.4.1, we abstract four design principles that realise positive values. Furthermore, in Section 2.4.2, we abstract four design principles that reduce negative values. These design principles are then combined to produce a set of eight **Ethical Platformisation (EP) Principles** in Section 2.4.3.

2.4.1 Realisation of Positive Values

Involving stakeholders in the design process directly aligns with a central tenet of *requirements engineering*, wherein the needs and constraints for a system are discovered and analysed in order to meet its stakeholders' goals. Consequently, requirements engineering becomes a socio-technical discipline for “developing requirements through an iterative, co-operative process” [35], [36]. Simply put, the people most qualified for characterising the needs of the stakeholders are the stakeholders themselves. In the context of social media platforms, where communities can come together for social-coordination, we propose that the community should solve their own requirements – this allows us to engage in co-engineering with them using a supporting platform. This is precisely the essence of a *public interest technology* (PIT), wherein technological potential can be leveraged to address the needs of stakeholders [29]. Accordingly, finding “solutions for the social good” directly translates to finding “solutions that involve stakeholders in the design process”, since from a community's perspective “social good” is subjective.

Beyond just consulting communities in the design phase, however, we must also consider their long-term trajectory. In order to foster lifelong engagement with a platform, we must first recognise that a community's requirements are **not static** [37]. By acknowledging that these requirements may change over time, we should acknowledge that a platform's functionality must *also* change over time, in order to continuously model and reflect the values of the stakeholder. This could

be termed requirements *re-engineering*, where outdated functions are superseded by new ones to address a dynamically changing goal.

A similar analogy exists in machine learning with *concept drift*, where the (statistical) properties of the target variable, which the model is trying to predict, change over time in unforeseen ways [38]. Consequently, the ‘predictive power’ of a classifier model may degrade if it is trained on historical data, as it will be unable to respond to the changing landscape of the real world [39]. Ultimately, an ethical platform is one that continuously evolves to enable the dynamic inclusion of stakeholders not merely during the design process, but through the entire operationalisation process. Accordingly, an ethical platform should support the ability for require runtime re-configuration.

When defining the characteristics of *third places*, Oldenberg drew a comparison to the role of a third place as a *leveller* [21]. Historically, the term *leveller* was used to denote anything that reduced men to an equality; the English coffee-house, for example, served to strip people of their classes and ranks in order to provide a neutral ground for interaction. Third places, then, were shown to incorporate and *expand beyond* this inclusivity, such that the content of person’s character was able to shine through, irrespective of their position in life. A similar argument can therefore be made for *digital* third places: they serve as (digital) levellers, whereby members of a platform-based community create identity and value from a starting point of social equality [40]. Reciprocally, if those members are aware of both the functional and normative aspects of ‘platform citizenship’, they can, and *should*, participate equally as well. Furthermore, digital platforms offer opportunities for dynamic inclusion. Similarly to how third places serve as a *home away from home* [21], such that people can come and go as they please, so too should digital third places create a sense of openness, thereby allowing voluntary association and dissociation and directly opposing the toxic creation of lock-in and digital dependence [41]. In serving as a (digital) ‘home away from home’, digital platforms should foster a sense of belonging and investment, helping to, in turn, create a sense of ownership. This sense of ownership, then, should serve as another incentive for equal participation and a strong disincentive for self-degradation of the digital third place.

2.4.2 Reduction of Negative Values

The adoption of *ethical platformisation* principles ought to redress the malicious side-effects stemming from the *platformisation* movement. Rather than the asymmetric power relations serving the platform instantiator over the stakeholders, ethical platformisation prioritises the ethical values of the community for which its platforms are intended to serve. Ultimately, this inverts the current

power relationship, as the platform provider stands to gain nothing, while the community stands to gain everything. For example, the *algorithmic disinformation* stemming from large-scale digital platforms results from the malicious intervention of AI systems and the inability for stakeholders to withdraw from their purview. By incorporating the *requirements re-engineering* principles, this allows for platforms to continuously evolve at the velocity of the community. This allows for the voluntary inclusion and removal of such AI systems, if the community withdraws consent.

Similarly, the harms associated with *surveillance capitalism* result from the unrestricted access of the platform providers to stakeholder data. By instead adopting the *ethical platformisation* design principle of first and foremost serving the community, stakeholders regain autonomy over their data.

Finally, the implementation of *addiction by design* stems from the financial incentive of platform owners to create user lock-in. Again, by prioritising the needs of the community over the needs of the platform provider, and maintaining an arms' length relationship between the two, there is no need for an attention economy; platforms generated by process of *ethical platformisation* mitigate these asymmetric power relations.

2.4.3 Ethical Platformisation Principles

Platforms, when leveraged bottom-up by a community, are fundamentally a framework for self-organisation. Communities utilise the platform to codify their knowledge into sets of logical rules. Through this lens, platforms become systems for the generation and enactment of rules used to govern the platform's actions. In *Kantian ethics*, the human will operates on the basis of subjective volitional principles called *maxims* [42]. Accordingly, *morality* translates to demands that apply to these maxims humans act on [42]. Kant continues by adopting the position that one should “act only in accordance with that maxim through which [they] can at the same time will that it become a universal law” [42]. This defines *Kant's Categorical Imperative*, and has been adapted by other academics to argue that *Kantian ethics* are not merely an abstract formalism, but provide a framework for moral reasoning [42], [43], [44].

In this vein, any platform rule or algorithmic policy can be seen as a *Kantian maxim* that is generated *ad hoc* by the community that operationalises it. Therefore, *ethical platformisation* might be thought of as the computational equivalent of *Kant's Categorical Imperative*, whereby the rules underpinning the generation of the platform *ecosystem* ought to be *universalisable* – the

mechanism for producing the platform ecosystem ought to only ascribe rules that could become universal law.

A platform ecosystem developed by process of *ethical platformisation* must therefore conform to a set of **Ethical Platformisation (EP) Principles**. Beyond redressing the malicious tendencies of conventional platformisation, *ethical platformisation* should also serve to re-empower communities through self-organisation mechanisms. We therefore abstract the following EP principles from the above discussion:

EP Principle 1 *Ground-Up Self-Organisation* – Communities should be able to self-organise from the *ground up*, thereby eliminating any external influence from the platform providers.

EP Principle 2 *Community Re-Empowerment* – Digital technologies have a tendency to entrench power asymmetries between stakeholder and provider [45]. In this context, communities may be re-empowered by EP to use platforms that enable them to exercise choice and control over their social arrangements [45].

EP Principle 3 *Legitimate Consent* – Legitimate consent demands that processes of voluntary association should be informed, meaningful and revocable [46]. Stakeholders that engage with a platform produced by EP should therefore be able to meaningfully give and revoke consent.

EP Principle 4 *Equal Participation* – Community members should be aware of both functional and normative aspects of ‘platform citizenship’, i.e., are functionally enabled to participate in administrative, governance and self-determination activities, and are aware that they ought to participate (in some sense) equally in such activities [47], [48].

EP Principle 5 *Multiple Economic Paradigms* – Some platform activity is essentially transactional, but those transactions do not have to be purely financially-oriented. Therefore, we propose the use of *Axial Currencies* [49], whereby the exchange of tokens takes place in economies of esteem [50], information [51], or favours [52].

EP Principle 6 *No Othering* – Asymmetric power relations created by the platform infrastructure enable platform providers to see platform stakeholders as inferior, creating a sense of ‘other’. Platforms produced under EP should therefore eliminate this power disparity to create community rather than division.

EP Principle 7 *Data Autonomy* – Platforms produced through EP should re-empower stakeholders with data subsidiarity and protection (e.g., end-to-end encryption). The

use of stakeholder data must reside at the level at, or ecosystem in, which it is generated. This mitigates the effects of *surveillance capitalism*.

EP Principle 8 *No Attention Economy* – Platform providers are incentivised to create stakeholder lock-in. This maliciously maximises platform engagement to commodify human *attention*. Platforms generated through EP must keep an arms’ length relationship from the provider to ensure that they serve the community first and foremost.

Scope and Non-Exhaustiveness

Each EP Principle is abstracted from a distinct observation of the ethics of platformisation. This entails that each principle either promotes the realisation of a positive value, or promotes the reduction of a negative value, per the ISO 7000 definition [9]. Therefore, while each principle is clearly motivated, it cannot be guaranteed that this set of eight principles is exhaustive; we would argue that this set is necessary and sufficient to the extent of our experience *so far*.

In practice, the introspection of a principle-driven framework is inherently open-ended. Alternative perspectives, experiences, or values could reasonably motivate the introduction of additional principles or the refinement of existing ones. As such, it is not possible to demonstrate that any fixed set of principles – including the set of eight presented here – is complete.

In a sense, the formulation of this set of eight principles is itself an adoption of *Kant’s Categorical Imperative* – a genuine desire for how we *want* ethical platformisation to be performed has resulted in the codification of a set of rules defining how ethical platformisation *ought* to be performed. This entails that the principles are deliberately universalisable: they could be reasonably applied at the platform ecosystem level, to inform the behaviour of *any* given platform, rather than tailored to a single instance.

Therefore, the conceptualisation of this set of eight EP Principles can be understood not as a definitive set, but as a *heuristic* for introspecting on the behaviour of platformised ecosystems with respect to ethical platformisation. Hence, the EP Principles are not merely descriptive, but *prescriptive*, expressing how platformisation *ought* to be conducted at an ecosystem level.

2.5 Platformisers

A singular, ethically designed platform is not ethical platformisation. Ethical platformisation denotes the process by which ethical values can be incorporated into an entire platform ecosystem. To achieve this process, an appropriate tool is required which we denote as a *platformiser*.

Simply, a *platformiser* is a tool that facilitates platformisation. If platformisation denotes the (increasing) penetration of platforms in different economic sectors and spheres of life [13], whereby the platform is increasingly becoming the dominant infrastructural and economic model of the social web [53], then the *platformiser* is the tool that permits the evolution of this ecosystem.

In the context of *operating systems*, the *Linux* architectural framework serves as a promising example of a platformiser. Through this lens, the core, ‘vanilla’ *Linux* kernel is analogous to a platform. A broader *Linux* ecosystem then emerges through the kernel driver APIs and subsystems, whereby third-party hardware vendors and driver developers release modules to augment the core platform. Individual *Linux* platforms then enter into this ecosystem by instantiating the core kernel and customising it from the ground up with loadable kernel modules extending the core *Linux* runtime. What results is an ecosystem of community-driven modules and provider-driven kernels that allow for ground-up self-organisation – stakeholders can opt into the *Linux* ecosystem by instantiating the core distributable as their operating system and subsequently customising it with bespoke functionality from the driver repository. Crucially, this architectural framework serves as a *platformiser* by facilitating platform evolution across an entire community of contributors rather than centralising it with a single platform owner; each kernel instance creates another ‘node’ in the ecosystem, and each downloaded module creates an ‘edge’ between that instance and that module in the repository. Accordingly, the *Linux platformiser* framework generates a (sparsely-)connected ‘graph’ of users, developers, and providers.

Platformisation was also introduced in the social-media sphere through the introduction of the *Facebook Development Platform*. This platform gave third party developers access to *Facebook* user data such as profiles, friends, and photos in order to extend the “Facebook experience” into external applications. This effectively transformed *Facebook* from a monolithic social media site into a connected ‘nexus’ of social media *platforms* via the introduction of a software development environment [53]. This *Software Development Kit* (SDK) permitted the evolution of an entire *Facebook* ecosystem, whereby the flagship *Facebook* site could now be augmented through the incorporation of third-party software. Furthermore, this architectural change to the platform

enabled the programmability of external websites through a third-party Application Programming Interface (API) [53]. In this way, the *Facebook Development Platform* can be reinterpreted as a kind of *platformiser* – this architectural model generates an ecosystem of community-driven, third-party modules to augment the provider-driven ‘core’ *Facebook* platform. Unlike with the *Linux* platformiser model, however, this *platformiser* creates a centralised platform ecosystem. Third-party add-ons serve only to augment the core *Facebook* platform, rather than locally-instantiated distributables.

Similarly, *Minecraft* exemplifies a platformiser in the video-gaming sphere that, through a community-driven effort, transcended *Mojang’s* initial conception of a monolithic, stand-alone game. *Minecraft*, in its initial release during 2009, was a multiplayer co-operative survival game requiring players to navigate a 3-D block world and mine for resources in order to survive against harmful mobs. With the release of robust plugin frameworks like *Bukkit* in 2011, however, augmentations could now be added to a *Minecraft* server without requiring players to install client-side mods. The current iteration of *Minecraft* as a server-generating tool and plugin-repository browser therefore serves as a *platformiser* – this architectural model has created an entire *Minecraft* ecosystem comprising the provider-maintained ‘vanilla’ *Minecraft* distributable, the developer-driven plugins and add-ons, and the user-driven multiplayer server instantiations.

In summary, a *platformiser* has no standardised form. In the context of ‘operating-system platformisation’, the *Linux* platformiser manifests as a distributable kernel and driver SDK. Similarly, in the context of ‘social-media platformisation’, the *Facebook* platformiser manifests as a core platform and third-party development API. Furthermore, in the context of ‘video-game platformisation’, the *Minecraft* platformiser manifests as a ‘vanilla’ base game and plugin repository. Ultimately, common to these three implementations is a mechanism for instantiating the (base) platform, and a mechanism for specifying third-party modules or plugins. Together, this coalesces users, developers, and providers into a single, symbiotic ecosystem that can be conceptualised as a connected graph of software ‘nodes’ linked by extension and instantiation ‘edges’.

Although a given *platformiser* may not satisfy the **EP Principles** out of the box, its generative capacity for producing a platform ecosystem can and *should*. For example, while the core *Linux* distributable may not address all ethical requirements, its capacity for extension and the generative potential of the *platformiser* enables community-provided modules to supplement the missing functionality. Ultimately, whichever principles are unfulfilled by the ‘vanilla’ platform instance can be fulfilled *ex post facto* through community-driven modules. Ensuring that such an ecosystem exists, wherein a core, extensible platform can be extended by third-party modules is

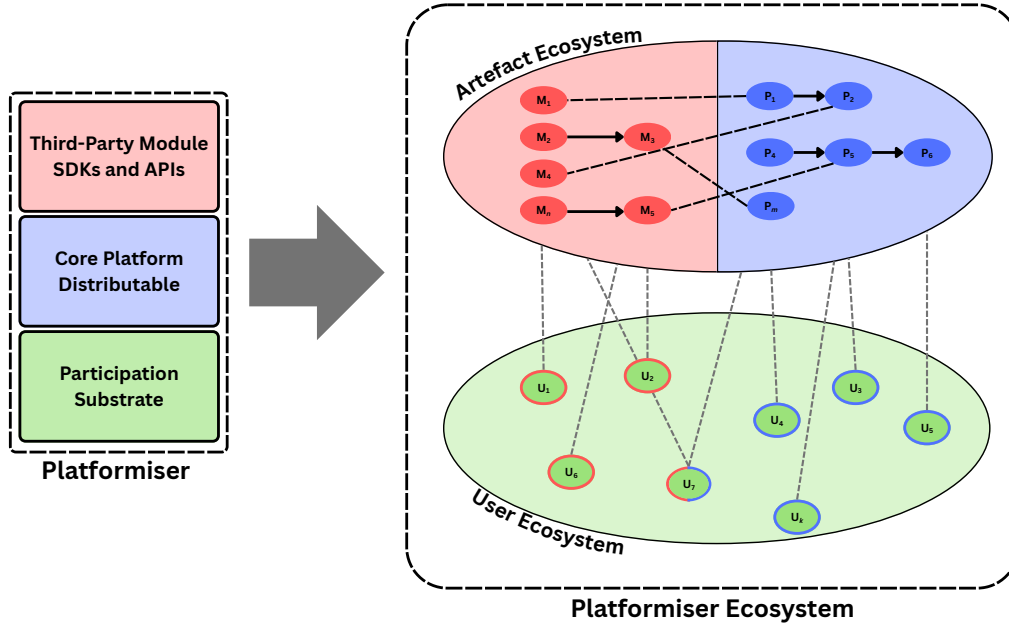


Figure 2.2: Role of *platformiser* in creating a *platformiser ecosystem*. This ecosystem is decomposed into two subsystems: an *artefact ecosystem*, consisting of third-party modules (red) and platform instances (blue), and a *user ecosystem* consisting of various user roles (green). The *artefact ecosystem* exhibits two dimensions: the vertical, *instantiation* dimension, wherein new artefacts are created independently of existing artefacts, and the horizontal, *innovation* dimension, wherein new artefacts are iterations of existing ones. Furthermore, the *user ecosystem* defines two roles; users can develop third-party modules (red border), join platform instances (blue border), or both (mixed border, U_7)

precisely the purpose of a *platformiser*, and achieved through its provision of SDKs, APIs and core distributables.

2.6 Summary

In summary, the evolution of the *digital platform* and subsequent proliferation by process of *platformisation* has created opportunities for disempowerment by malicious actors. Accordingly, we have defined the ‘counter-process’ of *ethical platformisation* along with eight **EP Principles** to explore the necessary conditions for how a platform ecosystem can re-empower the community for which it was designed, and oppose the abusive tendencies of platformisation. In order to guide the evolution of this ecosystem, however, we need an appropriate tool. This tool is called a *platformiser*, which we formalise in the following chapter.

3

Platformiser Formalism

Contents

3.1 Introduction	25
3.2 Abstract Platformiser Formalism	27
3.3 Platformisation: Architecture and Composition	31
3.4 Architectural Layer	32
3.4.1 Centralised Client-Server	33
3.4.2 Distributed Client-Server	34
3.4.3 Blockchain	35
3.4.4 Blocklace	37
3.4.5 Summary – Trade-Off Framing	38
3.5 Compositional Layer	40
3.5.1 ActivityPub	41
3.5.2 Microkernel Paradigm	42
3.5.3 Summary – Trade-Off Framing	43
3.6 Convergence – <i>PlatformOcean</i>	44
3.6.1 Technical Architecture	45
3.6.2 Socio-Political Substrate	47
3.6.3 Parallel Trust Streams	48
3.7 Summary	49

3.1 Introduction

The *Digital Platform Research Agenda* (DPRA [3]) defines the concept of a *digital platform* as the technical elements of software, hardware and associated organisational processes and standards. Beyond the platform is its *ecosystem*, defined by a collection of third-party complements to the core platform, including applications, modules, and micro-services. Furthermore, *boundary resources*,

including software tools and regulations, facilitate an arms' length relationships between the involved parties such as service providers, application developers, and end users. *Platform openness* can then defined by the extent to which these boundary resources support the complements within its operational life-cycle.

Establishing the extent of *platform openness* can be parametrised by such issues as accessibility, interoperability, and open-source software. These necessarily raise issues of how normative human values are embedded into technologies, and create ethical concerns. In Chapter 2, we have demonstrated that the monopolisation of large-scale global platforms at the application layer of the Internet creates opportunities for algorithmic misinformation [30], surveillance capitalism [54], and the AI-driven concentration of power [55]. Given that these effects have become endemic to the ecosystems generated by process of platformisation, redressing these issues necessitates a corresponding counter-process. This counter-process is ethical platformisation. Chapter 2 then continued to argue that in order to apply ethical platformisation not merely at a micro, platform level but a macro, *ecosystem* level, an appropriate tool is required called a *platformiser*. The role of a platformiser, then, is to guide and foster the development of platformised ecosystems with respect to the eight **EP Principles**.

Accordingly, this chapter specifies a formalism for one such platformiser called *PlatformOcean* at a component level. Section 3.2 supplies an abstract formalism of a platformiser with respect to six supra-functional design values and maps them onto a high-level component stack comprising a meta-platform, platforms, and modules. Section 3.3 then introduces how this high-level set of components can be manifested at a system level from the perspective of two orthogonal software dimensions: *architecturally*, through the choice of variably-centralised networking paradigms; and *compositionally*, employing either 'top-down' protocol-driven or 'bottom-up' microkernel-driven composability paradigms. Sections 3.4 and 3.5 then investigate these dimensions explicitly, providing a overview and set of drawbacks for corresponding technologies. Finally, Section 3.6 formalises the convergence of these two dimensions into a single, high-level system architecture for the *PlatformOcean* platformiser, and expands on the set of technical choices and socio-political assumptions required for its operationalisation.

3.2 Abstract Platformiser Formalism

Non-digital, industrial platforms presuppose strong modularisation while governed by an overarching design hierarchy [3], [16]. A direct analogy can be drawn to digital platforms, where the necessary *editability* and *reprogrammability* of digital technologies [3], [17], [18] can be achieved through a modular software architecture [4]. In this paradigm, a well-designed digital platform is analogous to a set of *Lego* bricks [4] – modular components of different shapes and sizes coalesce to form a single, cohesive, complex structure. With this formulation, users can readily separate and re-‘plug’ modules from/into the platform (*cf.*, *hot-plugging* [20]), as well as independently version external modules without the need for concomitant updates to the core platform [4]. This approach to modularity directly maps to the decomposition of a platform into its *core* and *periphery* components [2].

Since the *platformisation* movement denotes the process by which such platforms proliferate into the Digital Society [13], then *ethical* platformisation should denote the process of adopting the *IEEE 7000-2021* standard for ethical system design to facilitate and, most importantly, *guide* this proliferation. Central to this ethical standard is the involvement of stakeholders in the decision process, thereby codifying the ethical values of the communities that the platform is intended to serve. From this standard, this chapter identifies six design values to engineer a system from a techno-political perspective. To address the *technological* arm of techno-politics, we design a platform that supports:

- *Arms’ length relationships* – ensuring that any system component can be simplified to the extent that it only needs to maintain its own state and its interfaces to other components (*cf.* [3], [4]);
- *Runtime adaptation* – without service interruption, to address *concept drift* and continuous fitness-for-purpose in long-term community activity, where task performance and experience can develop over time [56]; and
- *Generative replication* – lowering barriers to entry by facilitating legitimate reproduction (e.g., by open source code and appropriate licencing practices) and variation that encourages generativity (i.e., unanticipated re-purposing) [57].

Furthermore, to address the *political* arm of techno-politics, we design a platform that also supports:

- *Stakeholder agency* – recognising one of Ostrom’s features of self-governing institutions [58], that those affected by rules should be included in the selection, modification and enforcement of those rules;
- *Decentralised governance* – avoiding getting ‘stuck’ with hierarchical structures that are no longer fit-for-purpose [37]; and
- *Public interest* – resisting the supposed *iron law of oligarchy*, which contends that any organisation is taken over by a clique that operate it to serve their narrow interest rather than a public interest or the common good (*cf.* [59]).

In particular, these six design values act as a *functional counterpart* to the eight EP Principles defined in Chapter 2. The EP Principles can be interpreted as a set of desired *suprafunctional* requirements for the entire process of ethical platformisation, that are only observable as an emergent property of the ecosystem. Conversely, the design values described above inform a *practical methodology* for how this specific platformiser should be built. In effect, the two sets operate at two tangential levels: the EP Principles apply to the process, whereas the platformiser design values apply to the tool that enables the process. Based on these design values, then, we extract three essential components for a platformiser to support ethical platformisation defined by:

Component 1 (Meta-Platform). *The meta-platform is ostensibly a ‘platform of platforms’. The meta-platform instantiates blank platform instances, stores and serves remote modules, and stores and serves pre-configured platform instances.*

Component 2 (Platform). *The platform is a software kernel that provides mechanisms for user registration, message passing, and dynamic module injection.*

Component 3 (Module). *A module is a remote piece of software that is dynamically injected into the platform to enhance its functionality.*

These components then form the basis for a modular system. The way this system addresses the techno-political design requirements defined above is shown in Table 3.1. The modular-system design paradigm has been shown to be disruptive in various arms of system design. In the inter-organisational arm, the traditional ‘top-down’ relationships for software development are instead replaced by arms’ length relations between app developers and platform providers [3], [60]. Furthermore, in the architectural arm, traditionally monolithic approaches are superseded by modular, ‘plugin-based’ architectures [3], [60]. This aligns with the definition of a “well-designed” platform

Design Value	Meta-Platform	Platform	Modules
Stakeholder Agency	Generates individual platform instances	Customisable by modules	Codify bespoke functionality
Decentralised Governance	Repository for governance modules	Customisable by modules	Codify governance arrangements
Serve Public Interest	Open-source tool	Values come ‘bottom-up’ from community	Ecosystem decoupled from provider
Maintain Arms’ Length Relationship	Decoupled from stakeholder	Decoupled from provider	Decoupled from provider and developer
Runtime Adaptation	Repository for modules	Dynamically reconfigurable through modules	Codify new functionality
Generative Replication	Repository for pre-configured platform instances	Encodable as array of module IDs	Encoded as dynamically-injectable code

Table 3.1: Cross-referencing of platform components with design values

ecosystem as an extensible codebase with third-party modules akin to *Lego* blocks [3], [4], [60]. Moreover, an analogy can be drawn to the ‘architecture-agnostic’ formalism of a platform as a low-variety *core*, and high-variety set of *periphery* [2]. In this model, there are a set of commonalities across all platform instances, governed by the static configuration of the *core*, which is customisable through (and differentiated by) the *periphery*.

Beyond offering benefits to the structure of the codebase, the choice of a modular architecture also directly re-empowers the platform stakeholders. Requirements engineering characterises the first phase of the design process, in which the specification for a platform can be formulated from informal ideas [35]. This process has been increasingly recognised as the key to improved software systems [61]. Designing the optimal platform then translates to involving the stakeholders in the design process, as requirements engineering is a socio-technical discipline for “developing requirements through an iterative, co-operative process” [35], [36].

A modular, plugin-based architecture directly supports this process. By offering users a minimal, static core, they can build a bespoke platform from the group up using the modular periphery.

The core then manifests as a micro-kernel which provides the lifecycle methods for the registration and de-registration of the periphery, which themselves manifest as external modules. This is precisely the essence of a microkernel architecture [62], [63].

Another important consideration in requirements engineering is identifying that a community's requirements are **not static** [37]. By acknowledging that these requirements may change over time, we should acknowledge that a platform's functionality must *also* change over time, in order to continuously model and reflect the values of the stakeholder. This could be termed requirements *re-engineering*, where outdated functions are superseded by new ones to address a dynamically changing goal. Again, this is directly supported by a plugin-based micro-kernel architecture. Allowing the external modules to be *dynamically* registered and de-registered is termed *hot-plugging*, a core functional requirement for plugin-based systems [20].

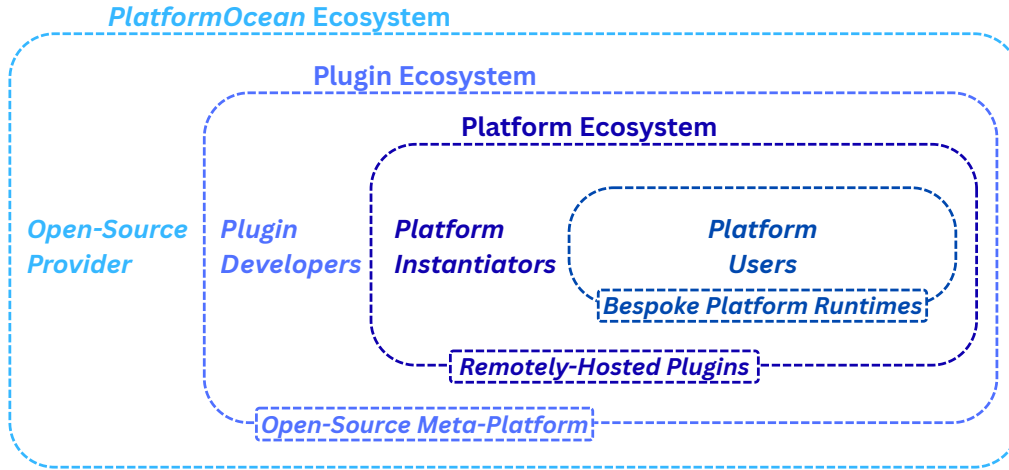


Figure 3.1: Embedded development ecosystems for the *PlatformOcean* architecture. Each layer represents a nested arms' length relationship. The corresponding *boundary resource* separating each nested layer is shown in *italics*, in the box sat on each boundary

The system architecture offered by this chapter sits one degree of abstraction beyond the modular platform architecture. Instead of having pre-existing platforms that can be extended, we define a mechanism for producing truly blank platforms that are built from the ground up. In this case, the complementarity/periphery are not a 'second-class' component (i.e., extending the functional core) but a 'first-class' component as they alone fully parametrise the platform. This promotes the designing and construction of bespoke platform instances, since communities can customise and develop their platforms from the ground up. Moreover, since the meta-platform instantiates a minimal core platform, this minimises the opportunity for the open-source provider to impose values onto the platform.

The *PlatformOcean* ecosystem leverages the open-source *meta-platform* to produce an arms'

length relationship between the open-source provider and the plugin ecosystem, such that platform instantiators are opaque to the meta-platform provider. Furthermore, the remotely-hosted plugins allow for a nested arms' length relationship between the plugin developers and the platform ecosystem, such that plugin installers are opaque to the plugin developers. Finally, the bespoke platform runtimes allow for the platform ecosystem to provide a final arms' length relationship between the platform instantiators and the platform users (*cf.*, [4]). In effect, then, each nested arms' length relationship is enabled and enforced through software tools and regulations, meaning that this software stack implements a set of *boundary resources* [3]. Accordingly, control is exercised at the layer interfaces, making ecosystem power architectural, not actor-based. This means that information processing design decisions, such as where computation occurs and where information persists, are made by those that are affected by those decisions. Figure 3.1 visualises these arms' length relationships and corresponding boundary resources.

3.3 Platformisation: Architecture and Composition

The widespread adoption of software platforms in the *Digital Society* forces us to reason about their technical infrastructure, and how this may be leveraged to create new forms of power and influence [18], [64]. Although certain implementations use these technologies maliciously to advance political gain, other implementations aim to oppose and counteract the *techno-politics* of platform proliferation.

The core concept to analyse with this intervention is *power*. In cyberspace, where *code is law* [65], it is postulated that the “control of code is power” [66], [67]. The critical question thus becomes “who shall write the software that increasingly structures our daily lives” [67]. In this section, then, we study the techno-politics of platformisation across two dimensions of implementation: an *architectural* dimension and a *compositional* dimension. For both dimensions, we identify the key technological decisions that underscore the possibility for techno-political *counter-disruption* – i.e., disrupting the original technological disruptors who used digital platforms to displace the traditional market economy, but have now themselves become entrenched. We conclude that the overriding motive for these alternative implementations is technology to support ‘user-centric’ public interest as the political imperative, over ‘company-centric’ private interest [68], in order to re-empower the individual.

Accordingly, the following sections explore the platform spectrum defined by Figure 3.2. Sec-

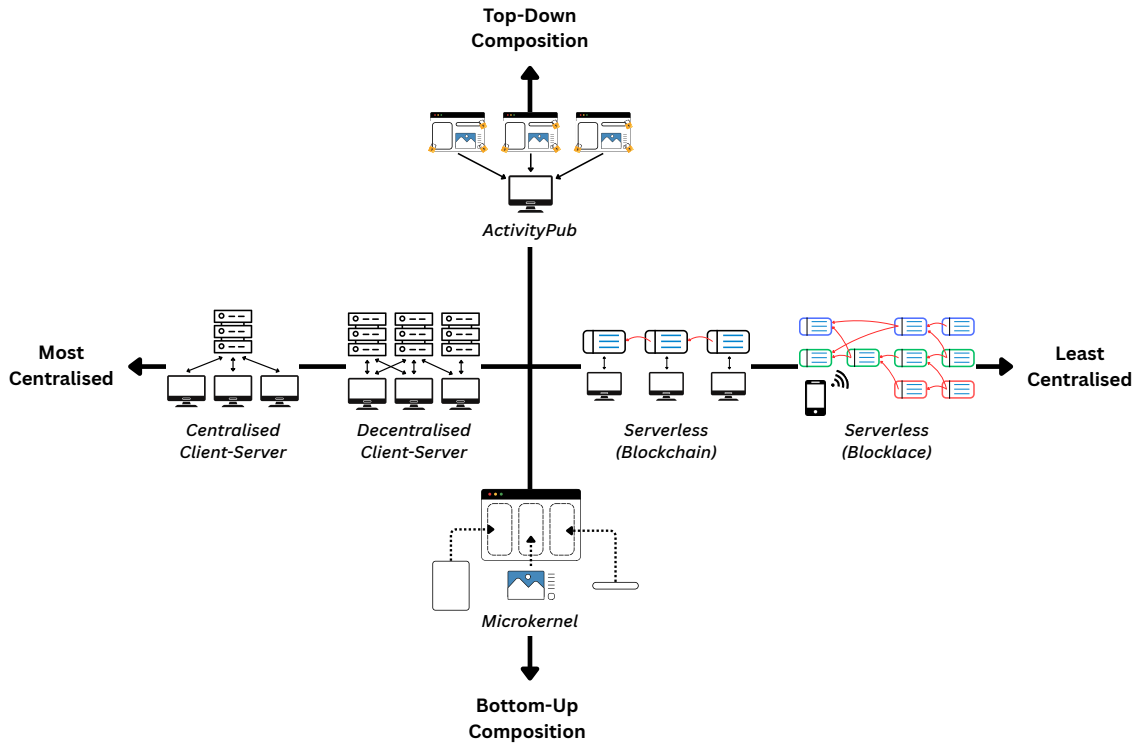


Figure 3.2: Spectrum of platform node architectures ranging from most centralised to least centralised, and composability paradigms from top-down (i.e., federating existing static software systems) to bottom-up (i.e., assembling dynamic software sub-systems)

tion 3.4 investigates the architectural ‘x-axis’ by surveying networking paradigms ranging from high to low centralisation. Symmetrically, Section 3.5 investigates the compositional ‘y-axis’ by surveying top-down and bottom-up composability paradigms. For both of these dimensions, we assess the paradigm by providing an overview and summarising the drawbacks associated with the design choice. Sections 3.4.5 and 3.5.3 propose the most viable architectural and compositional approaches, respectively, and their corresponding trade-offs. Finally, Section 3.6 combines the options derived in the previous sections to offer *PlatformOcean*: a distributed plugin architecture for ethical platformisation.

3.4 Architectural Layer

Instead of favouring additional technologies that sit on top of the pre-existing platform, an alternative intervention into techno-politics is the refactoring of the platform itself, as the choice of platform architecture directly impacts the distribution of power in the socio-technical system [69], [70] and directly implements ideological values [71]. This section explores four platform hardware architectures lying at different points along the power-centralisation spectrum, and the architec-

tural challenges posed by each. Figure 3.2 visualises this spectrum in the horizontal dimension.

3.4.1 Centralised Client-Server

Overview

The client-server network model is an architecture consisting of two components. The *server* generally functions as a remote database that stores data and runs programs that enable it to receive and process requests [72]. The *client* interacts with, and invokes methods on the server [73], in order to access these resources over a network connection [72], [74].

The *centralised* element refers to how, while there may be many users in the system (corresponding to multiple clients), all of the data storage and access methods are provided by a *single*, authoritative server [72]. The access relationship is therefore $1 \leftrightarrow N$, wherein N clients interact with 1 server.

The main appeal of a centralised client-server system is that the server acts as a ‘single source of truth’. This allows for a single, static server location (and corresponding network address), thereby simplifying resource sharing [72]. This centralisation paradigm can then be extended globally, as remote access can be granted to multiple client computers distributed geographically around the world [72], [75]. By identifying a single source of truth, and therefore a single point of failure, platform maintenance is simplified.

Drawbacks

In many ways, however, the centralised client-server architecture is directly responsible for the monopolisation of platforms at the application layer of the Internet. By centralising user data under a single authority (or company), the means of data ownership are deliberately opaque, and kept at arms’ length from the stakeholders [71]. This effect is exacerbated when the values of the shareholders are misaligned with the values of the stakeholders, for example when economic drivers inform the logic for digital technologies [54]. Users are therefore forced to relinquish their data autonomy and delegate trust to monopolistic actors.

3.4.2 Distributed Client-Server

Overview

The distributed client-server model aims to redress the issues of monopolisation enabled by the centralised client-server model by promoting less centralisation for its data flow. This architecture preserves the two principal components of the centralised counterpart – the *client* and the *server* – but reimagines their degree of autonomy.

Whereas, in the centralised case, a single server and multiple clients produces a $1 \leftrightarrow N$ relationship, the decentralised model promotes the use of *multiple* servers. These servers are all configured with common protocols and grammars such that any one client can interface with, and register itself to, any number of servers [76]. Accordingly, given M servers and N clients, this produces an $M \leftrightarrow N$ relationship.

The critical benefit afforded by the decentralised model is how it re-empowers users with *choice*. Instead of being locked into a single server (and, by extension, a single owner), users can choose which servers to join. Distributed server models are *autonomous*, such that they operate independently of one another with no power hierarchy. The centralised model enforces a single, top-down hierarchy wherein a single group of actors physically own all user data and users are forced to use their servers by engaging with the platform. Having the distributed model mitigates this problem by forming an *ecosystem* of servers; a *Byzantine* server instantiator harms only their local community, not the entire platform, since users can simply migrate away. This paradigm therefore enables users to have *legitimate consent*: it is both informed and, crucially, revocable [77].

A similar analogy exists with the *end-to-end argument*, stating that core functionality should be implemented at the endpoints of a communication system (i.e., platform), and not at a central level of the system itself [78]. A structural comparison can be made to distributed computing where, just like a central communication subsystem should not be overloaded with functionality [78], nor should a single, central server have total autonomy over the system’s communication pathways. This structural analogy also extends to fault tolerance. Distributing the number of points of failure means that, technologically, *Byzantine* agents at the platform endpoints do not compromise the functionality of the full ecosystem [79].

Furthermore, the modelling of multiple platforms as an ecosystem also gives rise to diversity. Instead of a centralised platform where a single group of actors dictates the norms and policies,

the capacity for any user to instantiate their own server means that they can run it with their own bespoke values. Any users who disagree with these values are free to leave, and not locked into the platform instance. This embodies the ‘see one, start one’ idea of distributed platforms [80]; users who are unable to find their ideal community can simply start their own by instantiating a server.

Drawbacks

The drawbacks of the distributed client-server model are twofold. Although no single entity has the ownership rights to the entire platform, each distributed server instance is still owned by an individual. This makes use of a platform *inequitable*, since the physical ownership of the technology instance enforces a hierarchy. Irrespective of the mechanisms in place, a server instantiator can still ‘unplug’ the system whenever they want. This problem is in part remedied by the discussion of legitimate consent above; users are not locked in to engaging with *Byzantine* (i.e., wilfully malicious) actors, and can migrate away from a server.

As well posing caveats to the clients in the distributed system, the host also absorbs risk when instantiating a server. In the centralised case, scalability issues are less of a concern; the monopolisation of a platform means that all resources can be funnelled into upgrading the single server instance. In the decentralised case, however, each server instantiator is solely responsible for its scalability. Given that this is unlikely to be a single, wealthy company, this becomes harder to support an increasing client-base.

3.4.3 Blockchain

Overview

A seminal intervention into the power asymmetries of techno-politics was the development of the *blockchain* architecture. This technology abstracts away the client-server dynamic to produce a truly serverless system in which components are connected as a network of peer-to-peer nodes.

Each node in the network helps contribute to a distributed state machine [81] by storing a reference to the state through a *ledger*. The ledger records which state changes have been performed in the past, in which order, and what the resultant state is. The ledger itself is comprised of a long chain of blocks, called the *blockchain*, each representing a group of *transactions* that performed some update to the global state [82].

Blocks in the blockchain are generated sequentially. Each new block contains a cryptographic hash of the previous block, thereby creating a sequential chain back to the *genesis block*. This models the blockchain as a linear *linked list* that is *tamper-proof* – if any one block is changed, so is its hash, meaning that all subsequent hashes no longer match [82].

The advent of blockchain was instantly disruptive to legacy networks and institutions of power, giving the opportunity to regenerate them [83]. The long-established client-server model, whether centralised or distributed, still ultimately placed the technology in the hands of an individual (or group of actors). Simply put, the ownership of a server is ultimately physical. If a user wishes to shut down the server, irrespective of the algorithmic rules put in place, the system will cease to run. The distributed model is able to shift this centralisation of power down to a subsystem/local community level, but the decentralised blockchain approach takes this a step further. If any single machine (i.e., node) in the system fails, there is no impact on the functionality of the overall system. This removes any autonomy on an architectural level, as no user is the ‘host’, no machine is a ‘server’, and no machine is the ‘client’.

This resolution of the architectural power asymmetry has promoted multiple techno-political benefits. The open representation of information has re-imagined and strengthened the legitimacy and transparency of big data [83]. Furthermore, the line between business and government has been made *porous*, thereby helping foster community trust [83], [84].

Drawbacks

The central drawback of the *blockchain* technology is its susceptibility to its decentralisation being undermined by large cryptocurrency holders, or *Crypto whales*. This stems from the two core consensus mechanisms that *blockchain* operates under to validate the blocks (and establish trust in a trustless network): *Proof-of-Stake* and *Proof-of-Work*.

In the *Proof-of-Stake* mechanism, the users that validate transactions are chosen based on their wealth [85]. This results in a *plutocracy*, where the cryptographic system is ruled by those with the most wealth [86]. Dangerously, these stakeholders are often early recipients of significant native tokens from pre-mining or centralised initial distributions [86]. Therefore, while technically decentralised, this system suffers from effectively the same problems as the centralised client-server architecture: users must effectively put their trust in wealthy actors who may not have views aligned with their own. Furthermore, this makes the system susceptible to the *Matthew effect*. If the validation of blocks, and therefore the earning of rewards, is tied to how much stake

is held, then the larger someone's staked holdings, the larger their return will be (or, colloquially, 'the rich get richer') [87].

A similar issue presents in the *Proof-of-Work* mechanism, regarding the tendency towards centralisation. In this mechanism, blocks are generated by solving a cryptographic puzzle; a number must be generated such that, when hashed with a block, the answer is less than a given target [88]. This process is called *mining*. The core issue with this mechanism is that performing these calculations is demanding of computational resources. *Bitcoin*, which uses *Proof-of-Work* as its consensus mechanism, consumes energy commensurate with entire industrialised nations [89]. This positions it, like other technologies such as Large Language Models (LLMs) and Data Centres, as a potentially substantive contributor to global warming [89]. Furthermore, the same issues of centralisation present themselves. The high cost of specialised mining equipment and electricity makes it difficult for small-scale participants to compete with large mining farms or operations. Similarly to the *Proof-of-Stake* system, this can result in a 51% *attack*, where if a single miner produces more than 50% of the work, they can theoretically manipulate transactions [90].

3.4.4 Blocklace

Overview

If the blockchain is implemented as a linked list of blocks, one other possible configuration is to connect the blocks in a (directed, acyclic) *graph* (DAG). This architecture is termed the *blocklace*.

In the *blocklace*, agents sign their own genesis blocks and form interlinked *virtual blockchains* [69]. Accordingly, each block may store any number of hash pointers to other blocks (unlike the single, previous pointer in the blockchain) to represent the edges in the DAG. Like the blockchain, this configuration is tamper-proof and non-repudiable, but delineates through ordering; the blocklace is partially ordered, whereas the blockchain is totally ordered [91].

The *blocklace* serves as the critical data structure for implementing the *grassroots* architecture, which was formulated to reduce opportunities for domination and control by ultra-high-performance server clusters in *blockchain* systems [91]. A distributed system is grassroots, if it can generate "autonomous, independently-deployed instances – geographically and over time – that can interoperate once interconnected" [91].

Grassroots systems seek to re-empower users through *digital sovereignty*, such that individuals

have full autonomy over their digital personae [69], [92]. Concerning digital communities, digital sovereignty is best expressed through the freedom of digital assembly, wherein people should have the opportunity to assemble digitally without the need for a third party [69]. In promoting digital sovereignty, grassroots directly combat the monopolisation of large scale, global platforms by removing the reliance on these third-party companies, thereby instead promoting political equality.

Furthermore, the blocklace addresses the energy concerns raised by the blockchain architecture. Given that consensus comes from digital signing as opposed to Proof-of-Work, nodes no longer compete to solve cryptographic puzzles. Instead, they cooperatively construct a shared *causal structure* of signed blocks, where agreement emerges naturally as nodes reference and acknowledge each other's blocks [91].

Drawbacks

The blocklace architecture does offer a compelling, if futuristic, alternative to centralised platformisation, in view of its 'democratic' (rather than 'plutocratic' (blockchain) or 'autocratic' (client-server)) approach to decision-making. However, sometimes, it is preferable to separate administrative from functional layers, especially in federated or socio-technical systems. This helps to establish a *zone of dignity* [77] that provides appropriate guardrails that constrain extremism and avoid infantilism. Moreover, 'democracy' is not necessarily the optimal governance regime for all situations (cf. [37]), and there is also a risk of degeneration into majoritarian tyranny, a completely different governance regime. Indeed, in support of structured forms of governance according to wisdom and knowledge, Plato argued against citizen participation due to a lack of understanding to make informed decisions.

3.4.5 Summary – Trade-Off Framing

Each architectural configuration provides their own strengths and weaknesses. In Table 3.2, we outline a theoretical implementation of how the platform components from Section 3.2 maps on to the different architectural frameworks.

Component	Centralised Client-Server	Decentralised Client-Server	Blockchain	Blocklace
Meta-Platform	Cloud	Cloud/CDN/P2P	On Chain	On Lace
Platform	Centralised Server	Decentralised Server	On Chain	On Lace
Modules	Cloud	Cloud/CDN/P2P	On Chain	On Lace

Table 3.2: Codification of platform components across different architectures

As the architectures become more decentralised, there are more opportunities for the components to be abstracted. If the remote modules can be modelled as functions, themselves encoded as pre-compiled code, there exists a natural trajectory from persistence on the cloud to dissemination via peer-to-peer. Furthermore, it is imaginable that these components can be embedded in an appropriate data structure and subsequently persisted on the blockchain or blocklace. A similar argument can be made for the meta-platform: given that this represents, reductively, a plugin and pre-instantiated platform repository, the backend can be implemented using the blockchain or blocklace data structures and served via a standard client.

Conversely, the more-centralised frameworks require a greater array of technologies to implement. In order to have a globally-accessible plugin and pre-instantiated platform repository, a service offering global resources is also needed. Although it is possible in the decentralised server paradigm to have multiple meta-platform instances, this produces a fragmented ecosystem, which inhibits the options for generativity offered by a global meta-platform. Accordingly, this requires either a centralised server or cloud implementation to avoid coupling with the platform instances.

Moreover, in order to preserve the various arms' length relationships and elements of digital sovereignty, the architectural approach must be conducive to platform operationalisation and maintenance. In particular, systems that are difficult to understand and maintain effectively create a *technocracy*, wherein only technically-literate 'experts' have the capacity to utilise (or worse, govern) the platform and its trajectory. This creates a significant barrier to entry for prospective platform users and massively compromises the platform's viability – any viable platform solution should be adoptable by a wide array of stakeholders, and should therefore be adaptable, maintainable, and generative.

Implementing an on-chain (or indeed, on-lace) platform solution is therefore a significant undertaking both in development and maintenance. The 'general-purpose' functionality of the modules transcends a standard block-based transaction, and would therefore require meticulously designed smart-contracts, for example. Furthermore, the codification of an on-chain platform raises privacy concerns: by default, blocks are permanent and transparent, however it may be favourable to represent the information shared over a platform ephemerally.

In conclusion, the most 'elegant' solution would be a single, abstracted implementation that is native to a single technology. The energy issues associated with consensus on the blockchain, and the guarantee of viability from blocklace, however, combined with a mutually high barrier to entry, make a decentralised approach unfeasible. The centralised client-server approach also

introduces problems of monopolisation, which are partially redressed by the decentralised client-server architecture. Therefore, this remains the most feasible paradigm to implement the platform discussed in Section 3.2.

3

3.5 Compositional Layer

A critical issue identified with most modern platforms is the issue of *interoperability*. In the pursuit of a thriving digital economy, for example, the EU commission charts a lack of platform interoperation as one of the most significant obstacles [93]. Architecturally, this has been caused by the lack of *modularity* [3], [4], in large-scale commercial platforms. A recommended design paradigm is to interpret as digital platforms as an extensible codebase, with third-party modules provided as a complementarity [3], [60]. Conversely, *monolithic* architectures enforce static codebases in favour of ease of development, testing and deployment, over the extensibility of *modular* architectures [94].

If a platform’s *architectural* layer defines the structuration of its core, distributed functionality, then the *compositional* layer equivalently defines how this can be extended with periphery to provide domain-specific functionality. The compositional layer therefore defines the techno-politics of the system across the two dimensions of *composability* and *compositionality*. Composability defines the *technological* way that a system is decomposed into smaller, independent modules that can be selected, assembled, and reassembled to satisfy different requirements. Compositionality, conversely, defines the *political* way that a system composes these independent parts into a collective, purposeful ensemble.

This section analyses two such composability paradigms. We begin by discussing *ActivityPub*, a protocol-based paradigm that provides a common grammar for the federation of existing platforms, and the *Microkernel*, a component-based paradigm that composes a shared runtime with dynamically-injected remote components. We term the composition of existing, available platforms a *top-down* approach, compared to the assembly of individual, single-function modules as a *bottom-up* approach. We visualise this difference in Figure 3.2 in the vertical dimension.

3.5.1 ActivityPub

Overview

A ‘middle’ ground between modular and monolithic architectures is the development of protocols to serve as a connective layer between existing monolithic architectures. This avoids forcing developers to refactor legacy code (i.e., migrating a platform to a different architectural implementation), and instead promotes interoperation via a common grammar that ‘sits on top of’ the codebase.

One such example is *ActivityPub* – a protocol and open standard for decentralised social networking. By providing both a client-to-server (C2S) API for creating and modifying content, as well as a federated server-to-server (S2S) protocol for delivering notifications and content to other servers, *ActivityPub* underpins the transition towards the *Fediverse* [68]. The *Fediverse*, a portmanteau of *federated universe*, is a paradigm that makes interoperation its primary concern, such that all platforms share a common language and can therefore interact (e.g., with *Mastodon*, *Pixelfed* and *BlueSky*). In this way, a datagram generated on platform i can be distributed to, and interpreted by platform j , irrespective of the underlying implementation details. This creates an ecosystem of architecture-agnostic platforms that all interact via a ubiquitous language.

The main benefit of *ActivityPub* is the promotion of decentralised control. This model is based on open collaboration between multiple, independent parties using an open-source grammar, over the filtering of information through undisclosed algorithms and unrestricted centralisation of sensitive user data [95]. By enabling decentralised control, users can curate a bespoke platforming experience by choosing servers that align with their own values and interests. Furthermore, platform trajectories are governed by stakeholders over shareholders – there is therefore no autonomy over the platforms, so no individual can change algorithms, ban topics, or delete communities. Accordingly, presupposed hierarchies (between follower and followee, say) are eliminated, giving more balanced behaviours between stakeholders [68].

Drawbacks

Central to *ActivityPub*, or indeed protocol-driven composition as a whole, is the requirement for all parties to conform to the same datagram grammar (i.e., protocol). Naturally, this limits per-instance customisability.

Furthermore, generativity is hindered by the high barrier to entry for protocol-supporting

platforms. Using this approach requires a deep knowledge of the federated model, which potentially excludes a large proportion of the user base.

3.5.2 Microkernel Paradigm

Overview

Microkernel, or *plugin*, architectures are a software design paradigm used to create extensible software through two main components. Firstly, a static *core* or *kernel* provides the ‘base set’ of functionality necessary for the system. This includes mechanisms for datagram distribution (i.e., message brokerage) and modular lifecycle management [62], [63]. Secondly, this kernel is extended with peripheral *modules* that ‘sit on top of’ the minimal kernel [62] to implement additional functionality. These modules are not directly compiled into the system, but connected via interfaces (or APIs) and other extension mechanisms [96]. Particularly in the context of network engineering, these modules are analogous to *plugins* or *micro-frontends*. They are independent from the core platform and are loaded dynamically (from a foreign repository, say) [20].

The microkernel architecture offers many benefits for the structuration of complex systems. Firstly, this paradigm gives a defined way to cope with the increased complexity of operating system development [62]. Drawing an analogy to operating systems, UNIX introduced the concept of a standardised operating system that is hardware independent [62]. By standardising this system at the kernel level, UNIX promoted portability, however struggled under extending scope as users wanted extensions to customise it [62]. This problem was tackled by the microkernel paradigm by providing a standard, generic minimum base that provides basic system services. These services could then be used by system builders to instead allow them to concentrate on creating bespoke system components [62].

Similarly, the microkernel architecture provides the compositional basis for the *micro-frontend* paradigm, wherein a website is decomposed into a ‘host’ (i.e., core application) and additional frontend features are modelled as injectable ‘micro’ frontends (i.e., modules). This enables different teams in a company to work on separate parts of a web application simultaneously [97], allowing for concurrent feature development and a streamlined workflow for AB testing.

Crucially, the microkernel paradigm directly gives the framework for developing bespoke, domain-specific systems. Defining a minimal core reduces the coupling between the kernel and its modules, thereby creating more opportunities for customisation. Furthermore, since the modules created un-

der this paradigm can be used in multiple different system environments, they become transparent and reusable [62].

Drawbacks

The main drawback presented by the microkernel architecture is the risk of fragmentation through module/plugin behaviour. This paradigm identifies the importance of a universal, generalised grammar to foster module-to-module interaction [62]. Allowing plugin developers to define their own ‘grammars’ or semantics, however, may create semantic incompatibility between separate platform instances. There is therefore no guarantee that two instances understand each other, even if they run the same kernel. This limits platform interoperability, and therefore federation.

Furthermore, since the modules are not compiled into the core system, but rather dynamically injected at runtime, this paradigm gives the possibility for Byzantine *remote code execution* (RCE). Plugins can introduce security vulnerabilities, dependency conflicts, or performance issues, and therefore require careful sandboxing and curation to maintain platform integrity.

3.5.3 Summary – Trade-Off Framing

In summary, the principal difference between an algorithmic, protocol driven approach and an architectural, platform driven approach to self-organisation is the *trajectory*. Protocols inherently place pre-existing platforms at the forefront, making this a ‘top-down’ approach – users must leverage the solutions provided by existing systems to create a bespoke platform. Conversely, instantiating new platforms inherently places pre-existing platforms at the background making this a ‘bottom-up’ approach – users can instantiate bespoke platforms to their specification from scratch.

The two approaches therefore delineate based on their customisability and capacity for domain-specific adaptation. By supplying a minimal core platform and promoting modularity, ‘bottom-up’ platforms can be tailored directly to user needs. In contrast, *ActivityPub* requires that all parties conform to the same grammar (protocol), thereby limiting per-instance customisability. The comparative evaluation of plugin- vs. protocol-centric models according to standard non-functional features is given in Table 3.3.

Note that promoting multiple ‘bottom-up’ platforms increases the possibility for fragmentation, and hinders interoperability and integration. By providing full autonomy of the customisation of

Feature	Plugin-Centric Model	Protocol-Centric Model
Extensibility	High, via plugins	Low, requires protocol extensions
Interoperability	Low unless formalised	High (by design)
Control	High (centralised runtime)	Low (fully decentralised)
Security & Policy	Easier to centralise and enforce	Harder to control uniformly
Evolvability	High (custom grammars per use-case)	Lower (needs consensus)

Table 3.3: Comparison of Plugin- and Protocol-Centric Models

a platform instance, users run the risk of invalidating the common protocols that would permit federation. There is therefore such a thing as ‘too much’ freedom. While this framework may re-empower users to create their own platforms and networks, they are still subject to the risk of technological ‘lock-in’. A platform built under technology x , given some common configuration through its kernel, may only be able to interface with another platform build under technology x . By using a common protocol, however, a mutual language exists across any platform.

The degree of decentralisation also comes with its own set of drawbacks. On one hand, the usefulness of a technology increases as its user-base increases [3], [98], [99]. Highly federated and interconnected systems therefore naturally promote the possibility for co-collaboration on separate platform instances, thereby increasing the user-base. On the other hand, a critique of excessive decentralisation was levelled by Pavel Durov (Telegram’s CEO) concerning ‘over-interconnectivity’: “the more connected and accessible you are, the less productive you are” [100]. This suggests that if users are overloaded with information-processing overheads, then this can detract from collective-action opportunities (*cf.*, [51]).

3.6 Convergence – *PlatformOcean*

Both the architectural and compositional dimensions of the platformiser necessitate a degree of *trust*. At a purely technological level, users must trust that the system satisfies a set of non-functional requirements, including resource guarantees, consistency, availability, and fault tolerance.

However, the operationalisation of a system – particularly a distributed system comprising multiple actors and independently operated components – also necessitates trust in the compliance with a set of socio-political assumptions. These include expectations around governance, ownership, and the behaviour of participating stakeholders, which cannot be enforced purely through technical means.

This section therefore introduces two parallel trust streams underpinning the *PlatformOcean* platformiser. The first concerns the technical architecture, which is derived following the above techno-political discussion. The second concerns the socio-political substrate required for the system to function in practice, encompassing governance structures, trust relationships, and norms of interaction across the ecosystem.

3.6.1 Technical Architecture

While the conventional approach to software design (or, indeed, any infrastructure) might begin with a solid foundation (i.e., *core*) and build upon it (i.e., with *periphery*) [2], we instead propose considering things the other way round. At the forefront of ‘good’ platform design sits modularisation [3], [4] and so, consequently, this should also manifest front and centre in the platform’s design principles.

We argue that modularity and, consequently, the capacity for domain-specific customisation lies not with the core, but indeed the periphery. In this vein, it is the mechanism for composability that is the primary concern, which the underlying platform architecture should support.

For the reasons argued in Section 3.5, we prefer a plugin-based architecture. This favours a ‘bottom-up’ approach where users don’t need to agree on highly-parametrised protocols, but can instead tailor behaviour to specific domains without modifying the core platform. This provides more opportunities for recombinant innovation [3].

Having established the composability paradigm, we now fit this plugin architecture onto a suitable platform configuration. As discussed in Section 3.4.1, centralising plugins and platform instances at the application level directly contributes to the monopolisation of global platforms, negating what this chapter aims to achieve. One alternative is the blockchain, discussed in Section 3.4.3, though this is undermined by its maintenance and operationalisation complexity, and susceptibility to centralised subversion. The blocklace architecture is an improvement which overcomes these limitations, but also makes a unitary commitment to a potentially degradable mode of governance, as discussed in Section 3.4.4.

This analysis therefore provides the basis for the design and implementation of *PlatformOcean*. The three abstract components used to support ethical platformisation are given a concrete implementation using a cloud/centralised repository for a *meta-platform* and a decentralised server for each *platform* instance. *Modules* are then manifested as *plugins* and are codified as remote

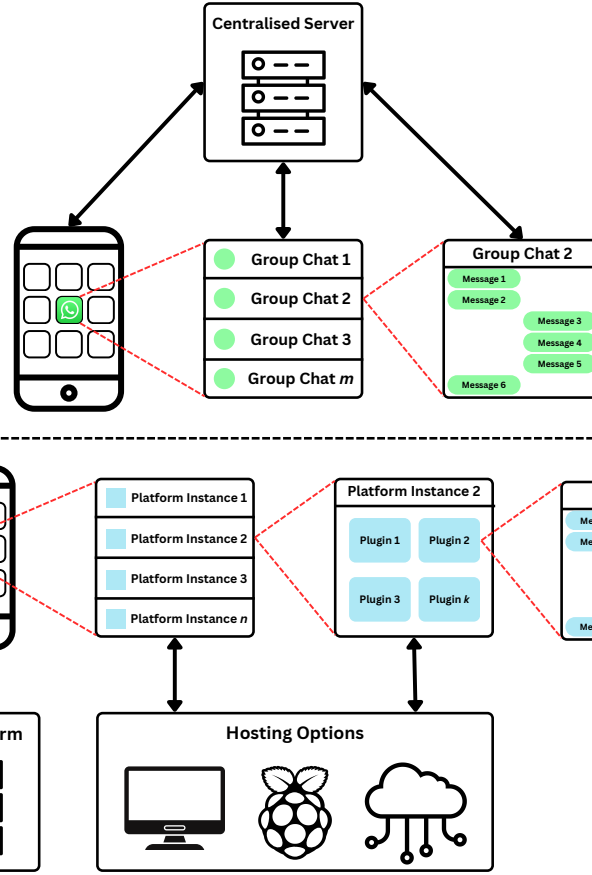


Figure 3.3: Complete abstract system architecture for *PlatformOcean* compared and contrasted with *WhatsApp*'s architecture. Note that *Plugin 2* implements a modularised group chat function (each chat is localised to the given platform instance)

modules stored on the *meta-platform*. These *plugins* can then be dynamically injected into each platform instance, thereby fulfilling the functional requirements of a *microkernel* architecture, and facilitating bottom-up composition.

We compare and contrast this architectural approach in Figure 3.3 with *WhatsApp*, a centralised, multimedia chat app, to visualise the interplay of these components. These approaches diverge in their degree of centralisation. This is highlighted in Figure 3.3 by inverting the position of the server in the *WhatsApp* and *PlatformOcean* stacks. In the *WhatsApp* architecture, the central component is a server handling instantiation of the app and the distributed message passing; all messages sent over the app are relayed (and stored) on a centralised server owned by the company. In contrast, in the *PlatformOcean* architecture, the only role of the centralised server is to provide a single point of contact for instantiating the server, client app, and plugins. Subsequently, all data is stored on, and all messages are relayed through, a localised platform server with multiple hosting options: for example a desktop, *raspberry pi* or on the cloud. The

following section formalises the *PlatformOcean* architecture by specifying the algorithms and data structures of each constituent component in turn.

3.6.2 Socio-Political Substrate

While, technologically, *PlatformOcean* provides mechanisms for ground-up self-organisation through its software, this relies on a necessary set of assumptions about the socio-political configuration of the technical stack and user-base.

In particular, two central assumptions are made. Firstly, in accordance with Ethical Principle 3, *PlatformOcean* enables legitimate consent, such that users may meaningfully withdraw from a platform. While, normatively, this ensures the possibility for *disassociation* from a given platform instance, it does not prevent users from forming *associations* with potentially malicious platform instances or operators with perverse incentives.

A naïve approach is to centralise the credentials of *PlatformOcean* users at the meta-platform level, thereby enabling a globally consistent registry for blacklisting malicious actors. While this is a possible solution for discouraging bad-faith operation, it introduces a strong dependence on centralisation and undermines per-instance generalisation.

By contrast, authentication to the platform level is more closely aligned with the principles underpinning the abstract *PlatformOcean* system architecture (see Figure 3.3). This preserves isolation between platform instances and allows users to maintain different identities across different communities. Socio-politically, however, this design choice shifts responsibility toward users and stakeholders, requiring them to exercise judgement in selecting with which platforms to associate. This entails that stakeholders must operationalise with respect to a consistent ideology: *don't join a platform that belongs to someone you don't trust*.

One possible mechanism to bridge the gap between centralised identity management and fully localised platform identities is the implementation of self-sovereign identity (SSI). In this paradigm, users are not defined by credentials issued and controlled by a central authority, nor are they confined to isolated, platform-specific identities. Instead, each user maintains a persistent, cryptographically verifiable identifier, through which they may selectively disclose attributes or credentials across platform instances [101]. This enables a form of portable identity and reputation without requiring a globally centralised registry. A similar approach was implemented with the *Open Mustard Seed* (OMS) [102], which also employs a decentralised identity paradigm that mediates trust

through user-controlled credentials rather than platform-imposed accounts. OMS demonstrates an early instantiation of these principles, relying on a “Trusted Compute Framework” and *RESTful* APIs rather than a fully decentralised blockchain approach.

Secondly, in serving as a plugin-repository, the meta-platform must maintain global correctness of the available plugin ‘catalogue’. If a plugin exists, then all stakeholders must know about it, rather than a subset that increases over time. Furthermore, functionally, the distribution of plugins ought to tolerate high throughput, fast retrieval, and guaranteed uptime. This implies a necessary centralisation of the meta-platform.

Therefore, this introduces possible concerns over the ownership model of the meta-platform. An actor with perverse incentives may introduce monetisation, say, to the plugin repository. Socio-politically, this necessitates that the ownership model of the meta-platform, while *technologically* centralised, has distributed ownership across multiple benevolent actors. One possible solution is to utilise a *trust* or *non-profit organisation* that is willing to supply the necessary meta-platform functionality without the incentive for any reward.

Alternatively, the reliance on a single meta-platform instance may be relaxed through a federated topology of independently operated meta-platforms, similar to a content delivery network (CDN). In this model, each instance maintains its own view of the plugin catalogue, acting as a local source of truth. Clients may then selectively associate with one or more such instances based on trust, effectively decentralising authority over plugin discovery. This approach reduces dependence on any single party, but necessarily sacrifices global correctness.

3.6.3 Parallel Trust Streams

In summary, *PlatformOcean* becomes contingent on two interrelated trust streams. Users engaging in the platform ecosystem must necessarily trust the technological architecture (i.e., ‘us’ as the software developers) with respect to a series of functional requirements, and trust the socio-political substrate (i.e., ‘them’ as the software consumers) with respect to a set of non-functional requirements. We summarise these parallel trust streams in Figure 3.4.

PlatformOcean comprises multiple nested strata. The meta-platform guides expansion at the ecosystem level, itself composed of multiple platforms that allow self-organisation at the community level, which are themselves composed of different combinations of plugins at the functional level. Across each of these nested strata, stakeholders place trust both *in* the technical artefact, and in

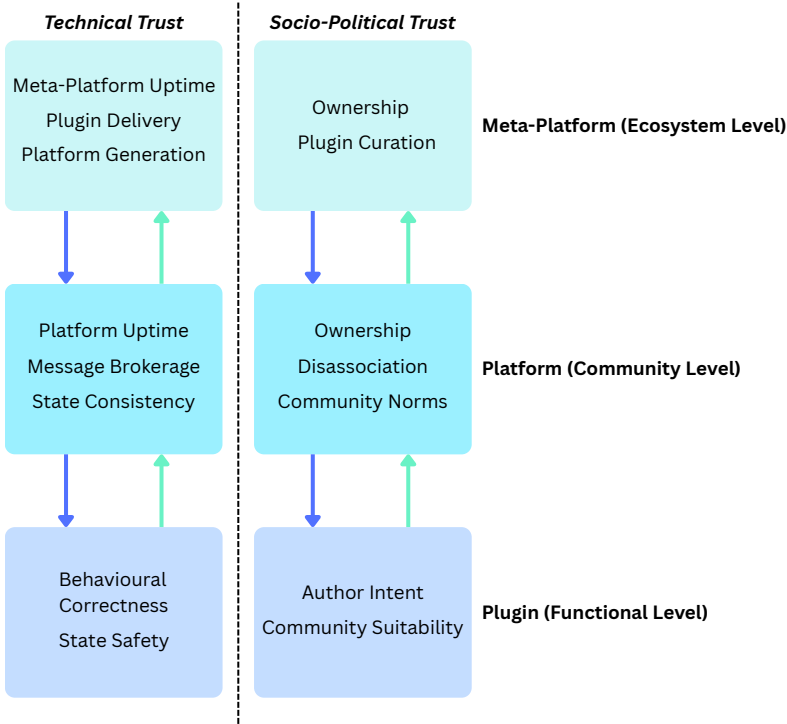


Figure 3.4: Technical and socio-political trust streams operating at the ecosystem, community, and functional level

the socio-political use *of* that artefact.

In effect, these parallel trust streams can be interpreted either as top-down or bottom-up in a kind of “chicken-and-egg” scenario. Stakeholders may initially place trust in the operation of the ecosystem as a whole, which subsequently encourages trust in participating within a platform and, by extension, in utilising its plugins. Conversely, trust may first emerge at the functional level through individual plugins, encouraging stakeholders to associate with or establish platforms, and ultimately extending trust toward the ecosystem itself.

Therefore, this platformiser entails that trust is recursive, trust propagates across strata, and legitimacy in this trust can emerge either top-down or bottom-up, depending on the perspective of the stakeholder.

3.7 Summary

In summary, this work can be seen both as a continuation in the advocacy of ethical platformisation, and as a continuation of the *Digital Platforms Research Agenda* (DPRA: [3]). The direction of this chapter has been from a techno-political perspective, such that the two key contributions are:

- an abstract platform formalism that identifies three core components (meta-platform, platform and modules) for satisfying techno-political design values (stakeholder agency, decentralised governance, serving public interest, arms' length relationships, runtime adaptation, and replication); and
- a critique of alternative platform infrastructures from architectural and compositional perspectives, concluding with a design paradigm to support platformisation based on a decentralised client-server architecture that preferentially satisfies these design values.

The significance of these contributions lies in the answer it gives to the DPRA's "unanswered questions in the digital platforms discourse". It is stated that:

The architecting of technology itself can to a higher degree be done through apps, micro-services and other modular elements rather than in monolithic arrangements, which raises the issue of how to architect these interrelationships [3, p.132]

This is, indeed, a central issue, and this chapter has mapped out the terrain for architecting the interrelationships between meta-platform, platform instances, and plugins (i.e., modules or micro-services). Furthermore, on the subject of openness, the DPRA goes on to claim:

[I]t is generally very difficult to see any openness being fostered for the main digital platforms. This raises a number of critical issues regarding the power dynamics and the direct impact of these digital innovations on everyday life, for example in terms of surveillance, labour market relations, and the distribution of wealth beyond a simplistic argument based on technical rationality [3, p.132]

It is evident that the monopolisation of monolithic digital platforms at the application layer has created asymmetric power relationships between service providers and service users. The proposal of *PlatformOcean* represents a distinctive pushback through the re-empowerment of previously disenfranchised people and communities [45]. This has a direct impact on the degree of difference and the degree of kind of platform that can be offered to stakeholders. Typical social media applications (*WhatsApp-like*, *Twitter-like*, *Facebook-like*) can be developed, but crucially each of these is owned and self-determined by the community itself.

We have therefore positioned *PlatformOcean* as a new class of software tool, which we term a *platformiser*. *PlatformOcean* has demonstrated that an alternative to privately-owned, monolithic

platforms is *specifiable* by formalising three high-level components for conducting ethical platformisation based on a plugin architecture. Functionally, *PlatformOcean* is a tool which performs this process of *ethical platformisation*, thereby generating platforms at their most basic level: vanilla, ‘blank’ instances that gain ontological meaning through ground-up customisation and operationalisation from the very stakeholders for which the platform is designed. Non-functionally, then, *PlatformOcean* performs this process, and therefore generates platforms with respect to, the eight **EP Principles** defined in Chapter 2.

However, it is not enough to simply assume the operation of this platformiser. The platforms that it produces must be able to implement ‘any’ *existing* platform, in order to enable a migration away from them. Chapter 4 therefore demonstrates that this platformiser is *computable*, and can generate platforms that are *platform complete*.

Platform Completeness

Contents

4.1 Introduction	54
4.2 Background	55
4.2.1 Requirements Engineering for ‘Social Good’	56
4.2.2 Plugin Architectures	57
4.2.3 Constructive Mappings	58
4.3 Universal Platform Formalisation	60
4.3.1 Functional Requirements	60
4.3.2 Domain Data, Logic, and Interface Components	61
4.3.3 Interaction Contracts	62
4.3.4 Non-functional Requirements	63
4.3.5 Formal Platform Tuple	63
4.4 Plugin-Architecture Formalisation	64
4.4.1 Platform Kernel	65
4.4.2 Plugins	65
4.4.3 Interaction Contracts	66
4.4.4 Formal Plugin-Architecture Tuple	67
4.5 Constructive Proof of Universality	68
4.5.1 Plugins as Bounded Contexts	69
4.5.2 Kernel-Plugin Contracts	70
4.5.3 Kernel for Non-Functional Requirements	72
4.5.4 Constructive Mapping and Completeness Corollary	73
4.6 Examples	74
4.6.1 Non-Functional Requirements	74
4.6.2 Functional Requirements	75
4.6.3 Interaction Contracts	76
4.6.4 Methodology	77
4.7 Summary	78

4.1 Introduction

Platformisation can be defined as the process underlying the emergence and proliferation of digital platforms [13], referencing any software infrastructure that promotes interaction in any sector of the *Digital Society*, including e-Commerce, e-Health, e-Learning, etc., but in particular social media. Social media platforms provide valuable prosocial services in the form of community inter-connection, knowledge exchange and collective action in the public interest [29]. However, platformisation at the application layer of the Internet has led from proliferation towards centralisation, where control over fundamental functions of social media has been concentrated in the hands of a small number of monolithic and monopolistic platforms. This has led in turn to surveillance capitalism [103], platform capitalism [104], the private ownership of the means of social coordination and an asymmetric distribution of power [105], constraints on marginalised content creators [106] and ‘acceptable’ platform governance research [107], and diminished stakeholder control over data, identity and functionality [108]. The social impact of this process has been to diminish multi-dimensional values-based interactions into a single dimension of price, where all interactions are perceived as transactional rather than relational.

Drawing an analogy with Ostrom’s institutional design principles for common-pool resource management by a community instead of by privatisation [58], normatively, there *ought* to exist an alternative to such privately owned and operated platforms. From a requirements-engineering perspective, one such alternative is a publicly accessible *plugin architecture* [62], [63], [109], [110]. This is a design paradigm that allows a platform to be constructed from the ground up, enabling direct stakeholder participation in defining, selecting, modifying and extending platform functionality (cf. Ostrom’s third institutional design principle of self-determination [58]). Plugins, or plugin architectures, manifest themselves in different ways across various software systems. For example, with the *Eclipse* and *VSCode* IDEs, plugins serve to transform the otherwise simple code editor into a sophisticated software development tool. Similarly, Web Browsers such as Google Chrome’s utilise browser extensions to enhance the search engine with quality of life features such as a dark mode, or dictation tool [96]. In addition, in the computer game *Minecraft*, plugins extend the functionality of servers, allowing for customisation of host features and gameplay mechanics.

However, for a plugin architecture to be a *viable* alternative to private platforms, as argued in [111], it must also be technically feasible. Specifically, the architecture must be *platform complete*, as per Definition 1:

Definition 1 (Platform Completeness). *An architecture is said to be platform complete if it is sufficiently expressive to realise, in principle, the full class of software platforms – whether empirically observed or theoretically conceived – within its structural and behavioural framework.*

In this chapter, to prove platform completeness of a plugin-based architecture, we use a constructive mapping approach as the basis for a refinement proof, i.e., there exists such a mapping which shows that a lower-level specification correctly implements a higher one [112]. We begin by formalising an abstract representation of a generic software platform as a tuple, grounded in established principles from *domain-driven design* (DDD) and microkernel architectures. We then define a constructive mapping from this abstract model to a plugin-based architecture, showing that for any platform expressible in the tuple formalism, there exists a corresponding configuration of kernel and plugin components realising its functionality. By explicitly constructing this mapping, rather than assuming its existence, we rigorously demonstrate that this architecture can implement any such platform while preserving isolation, modularity, and domain independence.

Accordingly, this chapter is structured as follows. Section 4.2 discusses requirements engineering for social media platforms that deliver a social good, and propose a publicly-accessible plugin architecture as an alternative to existing, private, offerings. Then Section 4.3 presents a universal platform formalisation, Section 4.4 presents a plugin architecture formalisation, and Section 4.5 gives the proof of platform completeness. This proof is based on constructing an algorithmic mapping between an abstract platform concept of Section 4.3 and the concrete plugin architecture of Section 4.4, in order to satisfy the requirement of “imagined in principle”. The proof is then grounded in Section 4.6 using examples of *Twitter-like*, *WhatsApp-like* and *Facebook-like* platforms, to illustrate and substantiate that a plugin architecture can satisfy the requirement of “observed in practice”. We conclude in Section 4.7 by arguing that this mathematical formalisation of platform abstraction and its refinement into a plugin-based implementation constitutes a novel intervention into the techno-politics of platformisation, and its potential social impact is to show that a *platform complete* plugin architecture can, in principle and practice, satisfy arbitrary community-centric, stakeholder-driven requirements.

4.2 Background

This section expands on the background to this work. It begins with a general survey of stakeholder engagement in Requirements Engineering, if the software product – in particular a platform – is

targeted at public interest or a ‘social good’. It then reviews the software design paradigm of plugin architectures as an approach to recombinant software development and innovation through its inherent modularisation, which supports stakeholder-centric requirements engineering. However, to ensure that a plugin architecture could, both normatively and functionally, implement any platform so engineered, we need a proof of platform completeness. The general approach to this – a constructive mapping – is summarised in Section 4.2.3.

4

4.2.1 Requirements Engineering for ‘Social Good’

The monopolisation of large-scale software platforms shifts the power to design critical features away from the stakeholder and towards the company owners. This creates an inherent tension between the pursuit of the ‘common good’ and the desire for profit [28]. In centralising this process, platforms often lack the civic engagement of the stakeholders themselves, causing software to be designed *for* users, and not *with* them [29]. Requirements engineering aims to address this issue by discovering and analysing the needs and constraints for a system in order to meet its stakeholders’ goals. Consequently, it is a socio-technical discipline for “developing requirements through an iterative, co-operative process” [35], [36].

Simply put, the people most qualified for characterising the needs of the stakeholders are the stakeholders themselves. In the context of social media platforms, where communities can come together for social-coordination, we propose that the community should solve their own requirements – this allows us to engage in co-engineering with them using a supporting platform. This is precisely the essence of a *public interest technology* (PIT), wherein technological potential can be leveraged to address the needs of stakeholders [29].

Accordingly, finding “solutions for the social good” directly translates to finding “solutions that involve stakeholders in the design process”, since from a community’s perspective “social good” is subjective. One example of an alternative technology to monopolistic platformisation is *ActivityPub*, which is a decentralised protocol for federated social networking [113]. By defining APIs for both server-to-server and client-to-server interaction, different platforms are allowed to interact forming what is (colloquially) known as the *Fediverse* [68]. This exemplifies PIT in practice by decentralising power, and promoting user agency and autonomy by allowing communities to host their own platform instances.

Furthermore, the client-server architecture itself has been criticised for directly contributing to *surveillance capitalism* through supporting platforms that use this as a business model [91]. To

combat this, the *Grassroots* architecture [69] is designed as a distributed system that can have multiple instances, independent of each other and of any global resources, that can interoperate once interconnected [91].

Beyond just consulting communities in the design phase, however, we must also consider their long-term trajectory. In order to foster lifelong engagement with a platform, we must first recognise that a community's requirements are **not static** [37]. By acknowledging that these requirements may change over time, we should acknowledge that a platform's functionality must *also* change over time, in order to continuously model and reflect the values of the stakeholder. This could be termed requirements *re-engineering*, where outdated functions are superseded by new ones to address a dynamically changing goal.

A similar analogy exists in machine learning with *concept drift*, where the (statistical) properties of the target variable, which the model is trying to predict, change over time in unforeseen ways [38]. Consequently, the 'predictive power' of a classifier model may degrade if it is trained on historical data, as it will be unable to respond to the changing landscape of the real world [39]. Ultimately, a platform may require runtime re-configuration, for which a *plugin architecture* is best suited.

4.2.2 Plugin Architectures

Plugin architectures, also known as *microkernel architectures*, are a software design paradigm used to create extensible software through two main components. Firstly, a static *core* or *kernel* provides the 'base set' of functionality necessary for the system. This includes mechanisms for datagram distribution and plugin lifecycle management [62], [63]. Secondly, this kernel is extended with *plugins*. These serve as independent, dynamically loaded modules that implement additional functionality. These plugins are not directly compiled into the system, but connected via interfaces (or APIs) and other extension mechanisms [96].

One example of a platform shaped by this architectural framework is *Minecraft*, a co-operative survival game where players can explore and interact with a block-based, 3-D sandbox world. As an online multiplayer game, *Minecraft* supports a client-server architecture where users can host a server instance and invite others to join. Though not strictly part of the core, 'vanilla' platform, third-party server distributions such as *Spigot* or *chapterMC* implement their own plugin APIs that enable the game's functionality to be extended. Server hosts can place external `.jar` files into a designated `plugins` directory, allowing additional modules to be dynamically loaded into

the server upon restart. Critically, this represents a **static** plugin architecture, as servers must be temporarily shut down in order to reflect changes to the plugin files.

Another example is *Figma*, a collaborative, browser-based interface design tool used for creating user interfaces and prototypes. *Figma* provides an official plugin architecture that allows developers to extend the platform’s functionality using *JavaScript* and the *Figma* Plugin API. These plugins operate within a sandboxed environment and interact with the design canvas via a restricted scripting interface, ensuring that extensions cannot directly modify the core runtime or compromise the shared workspace. Plugins can be developed, published, and distributed through *Figma*’s centralised plugin store, where the platform’s maintainers regulate submission, review, and visibility. Critically, *Figma*’s plugin ecosystem is powerful yet *domain-constrained* – unlike a fully generic plugin system, it is not possible to repurpose the platform for arbitrary computational tasks outside the domain of interface design.

From a requirements engineering perspective, plugin architectures provide a technological means to accommodate evolving stakeholder goals, since platforms built using this paradigm support runtime reconfiguration and dynamic extension [20], [110]. Furthermore, plugin development can become a co-operative process between the software developer and stakeholder, helping to produce ‘bottom-up’, bespoke platforms [109], [114]. By decentralising this control, plugin-based systems can help counteract the monopolisation of software design, aligning with public-interest values of integration, (civic) participation, and socialisation [29].

The critical issue, however, is that while a plugin architecture might *normatively* act as a generic development environment for community platforms, it may not be sufficiently *technically* expressive enough to reconstruct existing platforms, or to realise the full range of arbitrary community-defined requirements. In other words, it is necessary to determine whether a plugin architecture is merely conceptually inclusive, or formally sufficient to model any platform configuration in order to be *platform complete*, per Definition 1. The following section details the proof technique used by this chapter to prove that this architecture is sufficient.

4.2.3 Constructive Mappings

The proof technique we employ – constructing a mapping function from an abstract platform model to a concrete architecture – constitutes a *constructive refinement proof*. A refinement proof is a formal method used in computer science to prove that a lower-level, more detailed implementation correctly matches a higher-level, more abstract specification [115].

In software architecture research, this pattern is used in tools like *APML*, which enable architects to express and mechanically verify that concrete implementations preserve abstract architectural properties [116]. Here, the *soundness* of an *APML* proof is established by mapping it to a corresponding *Isar* proof in *Isabelle*.

Furthermore, *proof-carrying architectures* use *verified transformations* to guarantee that low-level architectural implementations are ‘correct’, relative to their high-level abstract descriptions [117]. These architectures define an *architectural hierarchy*, where high-level, abstract architectural descriptions are refined by mappings into more concrete descriptions [117], forming a hierarchy from abstract to low-level architectural elements.

A similar proof technique is also seen with *constructive data refinement*, whereby “abstract programs”, containing datatypes that are not normally available as primitives, are transformed into “concrete programs” by construction [118]. Therefore, similarly, abstract platform elements can be refined into concrete plugin implementations while preserving their structural and behavioural properties.

Mathematically, this aligns with the notion of *homomorphisms*, which are mappings between algebraic structures that preserve operations [119]. Model-theoretic results like the *homomorphism preservation theorem* further demonstrate the significance of these mappings in maintaining consistency and correctness across structural transformations [120].

Taken together, these approaches share a common paradigm: abstract specifications or high-level architectural elements are systematically mapped to concrete implementations while preserving correctness. This paradigm directly motivates our constructive mapping from abstract platform models to plugin-based architectures. Formally specifying this mapping function will allow us to conclude that there exists a direct transformation from any arbitrary platform instance to an implementation with this plugin-based architecture. This plugin-based architecture is therefore capable of implementing any possible platform instance, and is thus *platform complete*, per Definition 1. The following sections construct this mapping function in turn:

1. Section 4.3 formalises an abstract, high-level representation of a generic platform instance;
2. Section 4.4 formalises a concrete, low level representation of a plugin architecture; and
3. Section 4.5 formalises an algorithmic, constructive mapping function to refine the high-level platform representation into the low-level plugin architecture representation.

4.3 Universal Platform Formalisation

Modern software platforms are often complex systems composed of multiple interdependent features and services. To reason formally about their implementation, we propose an abstract representation of a platform as a three-tuple, using the principles of *domain-driven design* [121] in software architecture.

4

4.3.1 Functional Requirements

Domain-Driven Design (DDD) is a software paradigm that focuses on modelling software as individual *domains*, according to inputs from experts in the respective field. Central to the DDD paradigm is the proposition that each domain model not only can, but **must**, exist within a *bounded context*. This allows for the software to “explicitly define the context within which a model applies, [and] keep the model strictly consistent within these bounds” [121].

Critically, “a system can have multiple models, each in its own bounded context. Integration between bounded contexts requires careful translation and is explicitly defined” [121]. By then applying this paradigm to platforms, thereby interpreting a platform as a domain-driven system, we can model *the functional requirements* of each platform as a set of *bounded contexts*, where a bounded context defines the boundary within which a particular domain model is defined and applicable [121], [122]. Each bounded context represents a distinct *feature domain* of the platform’s overall operation: with *Facebook*, for example, this may encompass messaging, marketplace, and content feed. Moreover, bounded contexts have been shown to improve maintainability and system quality, and sustainability [122]. This gives a definition for platform decomposition:

Definition 2 (Platform Decomposition). *Any software platform’s functional requirements ($\mathcal{R}_f$) can be decomposed into a finite set of n feature domains ($\mathcal{C}$), where each feature domain can be implemented by a bounded context (C). That is,*

$$\mathcal{R}_f \equiv \mathcal{C} \tag{4.1}$$

$$\mathcal{C} = \{C_1, C_2, \dots, C_n\} \tag{4.2}$$

where, with this formulation, the functional requirements of a platform can be reinterpreted as cluster of domain objects treated as a single unit. Per the principles of DDD, this models a platform as an *aggregate* [121], [123].

4.3.2 Domain Data, Logic, and Interface Components

Having defined that any given platform functionality can be decomposed into a set of *bounded contexts*, this section aims to provide more granularity to the **composition** of a *bounded context*, in order to contextualise the component for platform engineering.

Evans also formalises *the building blocks of a model-driven design* [121], which defines the internal structure and behaviour of a *bounded context*. The core building blocks include:

- *entities* – objects with identity that persist over time
- *value objects* – immutable descriptors that carry attributes but no identity
- *services* – domain operations that don't naturally belong to an entity or value object

Within some bounded context C , particularly in the context of platform engineering, we can use these building blocks to abstract three key components in the “n-layer” web applications paradigm [124]:

1. **Data** (D): the ‘state’ associated with the bounded context, such as user data, messages, or posts (*cf.* the building blocks of entities and value objects);
2. **Behavioural Logic** (B): algorithms responsible for changing the data/state of the bounded context (*cf.* the building block of services);
3. **Interface** (I): APIs or user-facing endpoints that allow interaction with the context.

While the *data* and *behavioural logic* components have a ‘direct’ mapping in the context of platform engineering, the *interface* component may seem arbitrary. We reason that, given the visual nature of online software platforms, we must also consider the dimension of human-computer interaction. Instead of acting solely as isolated model instances (separating platform functionalities from each other), these instances must expose their context externally to the user in the form of an interface. This abstraction also aligns with the ‘original’ definition of a product platform as “a set of sub-systems and interfaces forming a common structure” [125], [126], [127]. This gives the definition for bounded context decomposition:

Definition 3 (Bounded Context Decomposition). *In the context of platform engineering, each bounded context (C) comprises **Data** (D), **Behavioural Logic** (B) and an **Interface** (I). That*

is,

$$C \equiv \langle D, B, I \rangle \quad (4.3)$$

4.3.3 Interaction Contracts

In accordance with DDD, all *bounded contexts* should remain isolated. In the generalisation to platform engineering, however, it is necessary for the contexts to interact (albeit through some common medium) so that information can be shared across feature domains. To facilitate this in the DDD paradigm, a *context map* is used. The purpose of a *context map* is to define (and visualise) the interplay between bounded contexts, such that it “charts the territory, giving the big picture of the contexts and their connections” [121]. Further developments of DDD have formalised these connections, devising three ‘categories’ of context-to-context interaction [128], [129], [130]:

- *partnership* – an “intimate” relationship between two contexts that cooperate to align their mutually dependent goals [130];
- *shared kernel* – a relationship when common parts of several contexts are extracted to another context/module to reduce code duplication [130];
- *customer-supplier* – a connection between two contexts, where one context (upstream) produces data, and the other (downstream) consumes it [130].

This is also aligned with the ISO42010 standard [131] that explicitly models interfaces and connectors as first-class architectural concepts.

To formalise this interoperability, *interaction contracts* are used to capture any rules, protocols, or channels through which the feature domain interacts with the rest of the platform. This is differentiated from the behavioural logic of a bounded context through scope: the behavioural logic allows for a direct translation of interface inputs into context data (and state updates), whereas the interaction contract allows for this data to be disseminated across a platform (if the contract deems it fit). We use the notation:

$$C_i \xleftrightarrow{X_{i,j}} C_j \quad \forall i, j \in \{1, 2, \dots, n\}$$

to denote an interaction between two bounded contexts C_i and C_j in $\mathcal{C}$ using an interaction contract $X_{i,j}$. By convention, we write $X_{i,j} = \emptyset$ to indicate that C_i and C_j do not interact. This gives an

$n \times n$ matrix of (nullable) interaction contracts for each pair of bounded contexts in $\mathcal{C}$ which is denoted by $\mathbf{X}$, where:

$$\mathbf{X} = (X_{i,j})_{i,j=1}^n \quad (4.4)$$

This fulfils an exhaustive description of all relationships between different contexts in the system. Again, per the principles of DDD, this interaction matrix is analogous to a *context map* [121], [128], [129].

4

4.3.4 Non-functional Requirements

Non-functional aspects of platform behaviour, such as authentication, consistency, and latency, are captured in a set $\mathcal{R}_{nf}$. These requirements apply globally across all bounded contexts, constraining both their internal behaviour and their interactions.

This abstraction is supported by the ISO25010 standard, which explicitly distinguishes functional and quality requirements [132], and the ISO42010 definition, which decomposes a software architecture into “its components, their relationships to each other and to the environment, and the principles guiding its design and evolution” [132], [133].

4.3.5 Formal Platform Tuple

Based on these principles, we can formally define a platform as follows.

Definition 4 (Platform). *A platform $\mathcal{P}$ is defined by a 3-tuple:*

$$\mathcal{P} = \langle \mathcal{C}, \mathbf{X}, \mathcal{R}_{nf} \rangle \quad (4.5)$$

where:

- $\mathcal{C}$ denotes the set of n bounded contexts (feature domains) that implement the functional requirements ($\mathcal{R}_f$) of a platform (as per Definition 2);
- $\mathbf{X}$ denotes the matrix of interaction contracts for these bounded contexts; and
- $\mathcal{R}_{nf}$ denotes the global, non-functional requirements.

This definition provides a high-level, language-agnostic representation of any practical platform in three parts: the *functional requirements* (through decomposition into bounded contexts), the *non-functional requirements*, and the *interactions* of the discrete platform components, thereby complying with the ISO42010 formalisation of a software architecture [133], while adding granularity to give a bespoke definition in the context of software platforms.

Trivially, by using Definitions 2 and 3, the platform three-tuple can be equivalently expressed as the five-tuple:

$$\mathcal{P} = \langle \mathcal{D}, \mathcal{B}, \mathcal{I}, \mathbf{X}, \mathcal{R}_{nf} \rangle. \quad (4.6)$$

which gives a more ‘data-driven’ interpretation of the otherwise abstract *bounded context* formalisation from the explicit enumeration of the building blocks from DDD.

4.4 Plugin-Architecture Formalisation

In the context of *platform theory*, platform architectures can be defined as “modularizations of complex systems” [2]. With this modular approach, platform systems are partitioned into two fundamental components: a *core* – comprised of components with *low variety* and *high reusability*; and a *complementary periphery* – comprised of components with *high variety* and *low reusability* [2]. These two component types are integrated through various interface specifications that ensure interoperability without sacrificing flexibility. This decomposition supports modular platform design by preserving a dependable core while accommodating diverse extensions and evolutions [2]. This section leverages this formalisation of a modular platform architecture to produce a formal model for a *plugin-based architecture*.

While the core/periphery decomposition of software platforms is traditionally expressed in terms of static system components, we may equivalently *reinterpret* this split as a *kernel-plugin* architecture. Here, the core functions as a minimal *kernel* that exposes stable interfaces and lifecycle primitives, while each peripheral component becomes a dynamically loadable *plugin*. This reinterpretation preserves the original low-/high-variety distinction yet introduces an additional degree of runtime flexibility: plugins can be added and removed from a platform dynamically without disrupting the kernel, thereby promoting reusability, extensibility, and the long-term evolution of the platform.

4.4.1 Platform Kernel

The *microkernel architecture* [62] (synonymous with a plugin architecture) is achieved by “implementing a simple core system that is extended by plug-ins that deliver additional system behaviour” [63]. This promotes an easily maintainable core with options for domain-specific customisation.

Furthermore, an otherwise-monolithic plugin architecture can be reimaged as a *distributed system* to provide dynamically-discoverable, self-hosted platform instances that can be customised for communities to self-organise. *PlatformOcean* [109], [110] abstracts the idea of a peripheral component into a *plugin*, and core component into a *vanilla platform*, providing the necessary lifecycle hooks and distributed datagram-sending methods to be customised with plugins. To accommodate the distributed system architecture, *PlatformOcean* also provides a *meta-platform* to act as a plugin repository, and a means of downloading/instantiating the separate platform instances.

This low-variety core provides only the fundamental services necessary for the system’s operation – such as lifecycle management, inter-plugin communication, and service registration/discovery – while delegating domain-specific functionality to the plugins themselves. This approach enhances flexibility, system stability, and ease of updates. This aligns with the definition of microkernel architecture as a flexible, modular structure with a minimal core system and independently developed periphery [62], [134].

From these implementations, the low-variety core or ‘vanilla platform’ can be abstracted, architecturally, as a single platform *kernel*, K .

4.4.2 Plugins

Plugins implement the high-variety, domain-specific functionality of the platform while remaining independent of each other and the kernel. By applying the DDD paradigm to an otherwise monolithic architecture, different domains of functionality can be reinterpreted as discrete modules [135], thereby forming a *Modular Monolith* [136], [137]. Accordingly, each of these discrete modules correspond to a bounded context, and can subsequently be implemented by (or reimaged as) a plugin, thereby encapsulating a specific feature domain and its associated logic, data, and interface requirements [121].

Communication between plugins is strictly mediated through the kernel’s interface mechanisms,

maintaining loose coupling and avoiding hidden interdependencies [63]. To enable extensibility and adaptability, plugins must support dynamic lifecycle operations: they can be added and removed at runtime without disrupting the kernel or other plugins [110]. This dynamic nature allows the platform to evolve continuously, supporting new capabilities while maintaining the integrity and stability of the core system.

4

A series of ‘quality-control’ functional requirements have also been developed for plugins. To align with the high frequency usage of a platform, plugins must be designed for maximal platform uptime, or equivalently, minimal platform downtime. Therefore, plugins must be added and removed from a platform interface without reloads or restarts, or otherwise long periods of inactivity. This concept is known as *hot-plugging* [20], [110].

Furthermore, given that plugins can be dynamically included in a pre-compiled system at runtime, this architecture presents the problem of remote/arbitrary code execution. Accordingly, by assuming that plugins are inherently *Byzantine*, restrictions must be put in place to prevent plugins from corrupting the hosting software or performing unintended crosstalk with other plugins. Restricting (and ideally immunising) this behaviour is achievable through *sandboxing* [20], [110], by loading plugins within a secure, bounding environment.

From these functional requirements, the high-variety peripheral components [2] can be modelled as a set of *dynamically loadable* microkernels (i.e., plugins), represented by the set $\mathcal{L}$.

4.4.3 Interaction Contracts

The kernel in a *microkernel architecture* serves as a minimal yet authoritative core [63] that provides the lifecycle methods for decomposing a system at runtime into constituent modules. These modules can be programmed in parallel, thereby shortening development time, and removing cross-cutting concerns, whereby editing one module affects another [134].

To fulfil these functional requirements, the kernel also provides a simple *interprocess communication facility* (IPC), which allows the subcomponents to call one another and exchange data independently [62]. This mechanism, critically, serves as one of the kernel’s *elementary services*, thereby providing a standard base which can support all other system-specific functions [62]. This paradigm is also applicable to real-time systems, where the kernel manifests as a simple, generic *executive* that supports specific real-time tasks.

Therefore, in a similar formulation to Equation 4.4, we use a kernel-driven *interaction contract*

$X_{i,j}$ to formalise the interoperation between two plugins, l_i and l_j giving the notation:

$$l_i \xleftrightarrow{X_{i,j}} l_j \quad \forall i, j \in \{1, 2, \dots, n\}$$

where, given n dynamically-loaded plugins, we formalise a similar interaction matrix $\mathbf{X}_K$ to denote the kernel-moderated set of $n \times n$ interaction contracts between the modular subcomponents in the plugin architecture.

4.4.4 Formal Plugin-Architecture Tuple

A platform architecture can be decomposed into its low-variety *core* and its high-variety *periphery* [2]. The core can be manifested as a *platform* [109], [110] or a general *kernel* (K). The periphery can be manifested as a set of *microkernels* [62] or *plugins* ($\mathcal{L}$) [109], [110]. By dynamically injecting these plugins into a supporting kernel, we formally define a plugin architecture as:

Definition 5 (Plugin Architecture). *A plugin architecture ($\mathcal{PA}$) is defined by the 3-tuple*

$$\mathcal{PA} = \langle K, \mathcal{L}, \mathbf{X}_K \rangle \quad (4.7)$$

where:

- K denotes the core, ‘vanilla’ platform kernel that provides runtime, communication, and lifecycle services;
- $\mathcal{L} = \{l_1, l_2, \dots, l_m\}$ denotes the set of dynamically loadable plugins (l) used to enhance the functionality of the core platform; and
- $\mathbf{X}_K$ denotes the matrix of interaction contracts mediated by the kernel to enable plugin-to-plugin interaction.

This definition is aligned with the principles of DDD. Associating the bounded contexts with the peripheral components and manifesting them as plugins isolates them from the platform core. This promotes a *decoupling* of the bounded contexts (i.e., feature domains), and unifies the functional requirements of *context isolation* (according to DDD) and *modularisation* (according to platform engineering [2]).

Crucially, this formalisation supplements the earlier, architecture-agnostic tuple in Equations 4.5 and 4.6 by providing a *constructive realisation* of a platform **architecture**, not simply the components. While the formalism of $\mathcal{P}$ captures a universal abstraction of a platform at the component level, the decomposition into core and periphery imposes an architectural style that can be directly instantiated. Moreover, it forms the basis for refining the conceptual model into a concrete implementation, which is the requirement for a refinement proof of platform completeness.

4

4.5 Constructive Proof of Universality

Having first abstracted a formalism for platforms, and a subsequent formalism for this plugin-based architecture, we prove that there is a constructive mapping between the two, i.e.:

Theorem 1. *There exists a constructive mapping f between an arbitrary platform representation $\mathcal{P}$ and a concrete plugin architecture $\mathcal{PA}$, such that:*

$$f : \mathcal{P} \mapsto \mathcal{PA} \quad (4.8)$$

Proof (sketch). The proof proceeds in several steps, each corresponding to the construction of one component of the mapping f .

Step 1 (Section 4.3 established the formal structure of the platform representation $\mathcal{P}$ and identifies its constituent elements.

Step 2 (Section 4.4) established the formal structure of the plugin architecture representation $\mathcal{PA}$ and identifies its constituent elements.

Step 3 (Section 4.5.1) demonstrates that a bounded context C existing on a platform $\mathcal{P}$ can be implemented by a set of plugins L in a platform $\mathcal{PA}$.

Step 4 (Section 4.5.2) establishes that an interaction contract $X_{c,c}$ existing on a platform $\mathcal{P}$ can be implemented using two *kernel-plugin contracts* $\langle X_{i,K}, X_{K,j} \rangle$ in a platform $\mathcal{PA}$.

Step 5 (Section 4.5.3) establishes that the non-functional requirements, $\mathcal{R}_{nf}$ comprising a platform $\mathcal{P}$ can be embedded into the common functionality of the kernel K existing on a platform $\mathcal{PA}$.

Step 6 (Section 4.5.4) formalises an algorithm for this mapping.

Hence, since every element of $\mathcal{P}$ can be mapped to a corresponding element in $\mathcal{PA}$, we conclude that such a homomorphic mapping f exists. $\square$

4.5.1 Plugins as Bounded Contexts

To satisfy the mapping, each element of $\mathcal{P}$ must have a corresponding implementation in $\mathcal{PA}$ by either the kernel or the plugins. Taking first a platform's functional requirements $\mathcal{R}_f$, we demonstrate how each a bounded-context feature can be implemented by a subset of plugins $\in \mathcal{L}$, such that $\mathcal{C} \mapsto 2^{\mathcal{L}}$.

From Definition 3, each bounded context of a platform can be decomposed into three characteristics: the **data** (D), **behavioural logic** (B) and an exposed **interface** (I). Critically, aligning with the principles of DDD, these components must be isolated from all other components, communicating only via a set of meticulously defined interaction contracts [121].

For example, Wolfinger provides a plugin architecture to sit on top of the .NET system [20] – an open-source developer platform. In complying with the .NET architecture, ‘hosts’ and ‘plugins’ are recontextualised as “slots” and “extensions”, respectively. In this paradigm, extensions (i.e., plugins) are implemented as traditional object-oriented classes that implement an interface defined on a slot. Therefore, the .NET plugins directly represent a bounded context as, in line with the principles of object-oriented programming, a class defines a clear boundary around its state and behaviour and exposes only a controlled public interface [138]. Moreover, this representation aligns with the domain-specific representation of a context in Definition 3: classes have some internal data, D , in the form of *fields* or *attributes*, some internal state-modifying business logic, B , in the form of *methods*, and an exposed interface, I , in the form of *getters* (though, in Wolfinger's formulation, the slot provides the interface for which the extension fulfils).

Ultimately, irrespective of the implementation medium, each plugin must maintain some internal state, apply state-modifying behaviours to evolve this state, and expose the resulting information either programmatically or through user interfaces. This mirrors the decomposition of a bounded context formalised in Definition 3. Consequently, if the functional space of any platform can be partitioned into disjoint bounded contexts, then it follows that the same partitioning can be realised in terms of dynamically loadable plugins. Therefore, we define context-plugin equivalence as:

Lemma 1 (Context-Plugin Equivalence). *Any bounded context (C) per Definition 4 can be reified*

by an equivalent set of one or more dynamically loadable plugins (L) per Definition 5, that is:

$$\forall C_i \in \mathcal{C}. \exists L_i \in 2^{\mathcal{L}}$$

4.5.2 Kernel-Plugin Contracts

4

The *Independent Plugin Principle* states that “plugins should be independent, with no dependencies on other plugins. The only dependency on the core system is through the plugin interface” [63]. In accordance with this principle, each bounded context (or plugin) should be independent, with no direct dependencies on other contexts. Communication between contexts is mediated exclusively through the core system. Accordingly, we can introduce two ‘proxy’ contracts via the platform kernel $K \in \mathcal{PA}$ that we denote the *kernel-plugin contract* to implement this communication while preserving the context-to-context isolation. This gives the definition for a plugin-to-plugin interaction contract as:

Lemma 2 (Plugin-Plugin Interaction Contract). *Any plugin-plugin interaction contract can be implemented using a kernel of equal or higher requisite variety and two sets of plugin-kernel interaction contracts, thereby preserving isolation between plugins. This is expressed as:*

$$L_i \xleftrightarrow{\mathbf{X}_{i,j}} L_j = \langle L_i \xleftrightarrow{\{X_{l,K} | l \in L_i\}} K, K \xleftrightarrow{\{X_{K,l} | l \in L_j\}} L_j \rangle \quad (4.9)$$

where:

$$\mathbf{X}_{i,j} = \{X_{i,j} \mid l_i \in L_i \wedge l_j \in L_j\} \quad (4.10)$$

Therefore, the kernel provides an intermediate route for the information flow, between plugins, such that for any given contract:

$$\mathbf{X}_{i,j} \equiv \mathbf{X}_{i,K} \circ \mathbf{X}_{K,j} \quad (4.11)$$

where the $\circ$ operator denotes a composition of interaction contracts.

Lemmas 1 and 2 give the following equivalence:

$$C_i \xleftrightarrow{X_{i,j}} C_j \equiv L_i \xleftrightarrow{\mathbf{X}_{i,K} \circ \mathbf{X}_{K,j}} L_j \quad (4.12)$$

which therefore provides the following mapping:

$$\mathbf{X} \mapsto \mathbf{X}_K \quad (4.13)$$

This denotes that any communication pathway between bounded contexts, themselves realised by plugins, is replicable with an intermediary pathway through a kernel. Critically, the kernel must be sufficiently *expressive* to handle the information conveyed by the interaction contract, and have sufficient opportunities for horizontal scaling to handle the latency constraints. For example, the kernel must be able to parse and store the datagram relayed by the contract. Therefore, this contract is essentially being fulfilled by a kernel of equal or higher requisite variety [139] than the plugin(s) it mediates.

A similar approach is implemented with *PlatformOcean*, where plugin instances do not communicate directly, rather the kernel defines both a plugin protocol for dynamically routing information via a *pluginKey*, and kernel-plugin contracts for subscribing to a websocket instance [109], [110]. In this way, the kernel acts as an interlocutor/message broker that exposes routing mechanisms for mutual websocket subscriptions, over a direct pathway for mutating plugins.

Beyond a *kernel-plugin contract* serving as a intermediary path for facilitating plugin-plugin (i.e., context-context) interactions, this contract also provides, for example:

- Lifecycle hooks exposed by the kernel;
- Message/event subscription and publication mediated by the kernel;
- Service registration or discovery through the kernel.

Thus, the *kernel-plugin contract* represents all the ways a plugin (i.e., bounded context) can interact with the main platform while preserving independence from other plugins. This ensures that the platform remains modular, extensible, and fully compatible with microkernel design principles.

This elevated responsibility of the kernel is further aligned with the representation of a platform's functional requirements in Section 4.3.1. By considering these functional requirements as a set of feature domains (realised by bounded contexts), we have reinterpreted the platform as an

aggregate, wherein these discrete contexts function instead as a single unit. The role of the kernel, then, is to serve as the *aggregate root* [121], [123], acting as the main entity that ensures consistency within the aggregate [123]. In serving as an interlocutor for the communication pathways (and holding elevated privileges to mediate these communications), the kernel can also be considered as codifying the DDD *context map* [121], [128], [129].

4

Furthermore, the kernel serves the purpose of a DDD *repository*, since it manages the retrieval and persistence of domain objects (*cf.*, the lifecycle methods of plugins) [121], [123]. Through the implementation of plugin-kernel contracts, the kernel also abstracts data access concerns [123], further solidifying itself as a kind of repository object.

4.5.3 Kernel for Non-Functional Requirements

In addition to mediating functional interactions between plugins, the kernel (K) is responsible for enforcing the platform's non-functional requirements (R_n). These requirements encompass *globally operational* properties such as security, fault tolerance, consistency guarantees, resource allocation, and performance constraints [132], [140]. Particularly in the context of platform engineering, these entail authentication and consistency.

By centralising R_n at the kernel level, the architecture ensures that each plugin remains agnostic to cross-cutting concerns while remaining compliant with platform-wide policies. That is, if the platform kernel is responsible for all plugin injection, it can also enforce *hot-plugging* and *sandboxing* [20], and retain autonomy as an interlocutor for all communication pathways. Formally, for each plugin $L \in \mathcal{L}$, all plugin-plugin interactions are mediated by the kernel, which enforces $\mathcal{R}_{nf}$ as invariants across all communication pathways and runtime behaviours.

This design aligns with the microkernel architectural principle of a minimal yet authoritative core [63], wherein the kernel provides a small set of services that govern both plugin interaction semantics (with either the kernel or one another), and operational constraints such as lifecycle mechanisms or access to public resources (e.g., kernel-managed memory) [62]. Crucially, this enables the periphery to evolve *independently* from the core system without compromising global correctness or quality attributes [134].

A similar workflow is seen with the *micro-frontend* paradigm, where a website is decomposed into a 'host' (i.e., core application) and additional frontend features are modelled as injectable 'micro' frontends (i.e., plugins). This enables different teams in a company to work on separate

parts of a web application simultaneously [97].

One example of a non-functional requirement that can be embedded into the kernel is security and data provenance. Given that the kernel is the ‘invariant’ (i.e., common to a platform and independent of the plugins), an authentication layer can be attached here. For example, *PlatformOcean* implements the kernel as a type of *state machine*, such that it is impossible to join a platform instance (nor interface with any plugins) until relevant user authentication has been performed [109], [110].

4.5.4 Constructive Mapping and Completeness Corollary

If the mapping f exists, there should be an algorithm to compute it. We complete this section by giving such an algorithm in Algorithm 1.

Algorithm 1 Constructive mapping from abstract platform to concrete plugin architecture

Require: $\mathcal{C} = \{C_1, \dots, C_n\}$ (bounded contexts), $\mathbf{X}$ (interaction contracts), $\mathcal{R}_{nf}$ (non-functional requirements)

Ensure: K (platform kernel), $\mathcal{L} = \{l_1, \dots, l_m\}$ (loadable plugin set extending the platform kernel), $\mathbf{X}_K$ (kernel-driven interaction contracts)

```

1: Instantiate  $K$ 
2: Configure  $K$  with policies from  $\mathcal{R}_{nf} \in \mathcal{P}$  (authentication, availability, persistence)
3: for each bounded context  $C_i \in \mathcal{C}$  do
4:   Create plugin subset  $L_i \in 2^{\mathcal{L}}$  per Definition 1
5:   Implement  $D_i$  as plugin-private storage (local state, allocated kernel memory)
6:   Implement  $B_i$  as plugin-internal state-operating functions, using only  $K$ 's storage and mes-
     saging primitives
7:   Implement  $I_i$  via  $K$ 's interface hooks (HTTP endpoints, event listeners, etc.)
8: end for
9: for each plugin subset  $L_i$  do
10:   Load  $L_i$  dynamically and inject into  $K$ 
11: end for
12:  $\mathbf{X}_K \leftarrow \{\}$ 
13: for each interaction contract  $X \in \mathbf{X}$  do
14:   Decompose  $X$  into the set of two kernel-plugin contracts  $X_K$  per Lemma 2
15:    $\mathbf{X}_K \leftarrow \mathbf{X}_K \cup X_K$ 
16: end for
    return  $\langle K, \mathcal{L}, \mathbf{X}_K \rangle$ 

```

Corollary 1 (Platform Completeness). *Given Algorithm 1, it follows that any well-formed platform representation $\mathcal{P}$ can be implemented as a concrete plugin-based architecture $\mathcal{PA}$ while preserving the compositional and behavioural relationships between its components. Therefore, the plugin architecture is platform complete, in accordance with Definition 1.*

Our proof constructs (f) explicitly, demonstrating that every bounded context ($C \in \mathcal{C}$) is realised by a plugin ($L \in \mathcal{L}$) preserving functional, behavioural, and non-functional properties. This

confirms that our plugin architecture is expressive enough to implement any platform described in the abstract model. This *homomorphic* function maps an **abstract model** to a **concrete architecture** while preserving semantics, and relations between the platform components.

4.6 Examples

4

In this section we demonstrate how the platform formalism can be implemented in practice, using three example platforms that expose two dimensions of increasing implementation complexity: a *Twitter-like* platform, comprising a single, (homogeneous) feature domain; a *WhatsApp-like* platform, comprising multiple homogeneous feature domains; and a *Facebook-like* platform, comprising multiple heterogeneous feature domains (*cf.* [69]). We illustrate this in Table 4.1.

		Feature Variety	
		Homogeneous	Heterogeneous
Feature Cardinality	One	<i>Twitter-like</i>	X
	Many	<i>WhatsApp-like</i>	<i>Facebook-like</i>

Table 4.1: Characterisation of example platforms across two dimensions of complexity

We define explicitly the *feature domains* that constitute these platforms, as well as an example of the *context-context interactions contracts* that these platforms may use. In the context of social media platforms, particularly with the subset that we have chosen, there is substantial crossover in their non-functional requirements. For brevity, and to avoid redundantly repeating these requirements, we enumerate a possible $\mathcal{R}_{nf}$ in the following section:

4.6.1 Non-Functional Requirements

A theoretical $\mathcal{R}_{nf}$, using examples from literature [141], [142], is:

- High availability – the brokering between sending and receiving messages should have a high uptime;

- Consistency – there should be a synchronisation of message / data states across various users. A platform accessed on machine a should (eventually) be identical to a platform accessed on machine b ;
- Scalability – the system should remain functional even with increasing load (space complexity) from users and messages; and
- Persistence – user and message data should be *durable*, such that it persists without corruption in a shared repository

Having defined an example set of non-functional requirements, $\mathcal{R}_{nf}$, common to these social media platforms, the following sections explore how we can specify both the functional requirements, $\mathcal{R}_f$, and a theoretical set of interaction contracts, $\mathbf{X}$, to formally specify a platform in the form $\langle \mathcal{C}, \mathbf{X}, \mathcal{R}_{nf} \rangle$.

4.6.2 Functional Requirements

Twitter-like

The *Twitter-like* platform represents the simplest platform implementation. Reductively, this platform must enable users to ‘blog’ by sending many-to-many datagrams comprising text or images across a network. The functional space of a *Twitter-like* is then a kind of open *group chat*, where users send publicly-accessible messages across an individual shared chat window:

$$C_{\text{groupChat}} = \text{groupChat} \qquad \mathcal{C} = \{ C_{\text{groupChat}} \}$$

WhatsApp-like

The *WhatsApp-like* platform can be considered as a coalescence of multiple separate *Twitter-like* platforms. Rather than a single shared group chat, the *WhatsApp-like* can be implemented through m distinct group chats that each user can register to a subset of:

$$C_{\text{groupChat}} = \{ \text{groupChat}_i \mid i \in \{1, \dots, m\} \} \qquad \mathcal{C} = \{ C_{\text{groupChat}} \}$$

Facebook-like

The *Facebook-like* platform is the most complicated example due to the variety of functionality it provides. Accordingly, this platform's features space comprises multiple, heterogeneous elements. Firstly, a *Facebook-like* platform provides standard peer-to-peer messaging in the form of the 'messenger' application. This can be implemented through m group chats. Furthermore, users also have access to a personal profile, or 'wall', where they maintain a list of friendships, personal information, and localised posts. Given m users, we can consider n profile components. Finally, a *Facebook-like* platform offers a marketplace feature, where users can exchange goods and services. This can be considered as p 'shop' components – one for each item – that multiple users can engage with:

$$\begin{aligned} C_{\text{groupChat}} &= \{ \text{groupChat}_i \mid i \in \{1, \dots, m\} \} \\ C_{\text{profile}} &= \{ \text{profile}_j \mid j \in \{1, \dots, n\} \} & \mathcal{C} &= \{ C_{\text{profile}}, C_{\text{groupChat}}, C_{\text{shop}} \} \\ C_{\text{shop}} &= \{ \text{shop}_k \mid k \in \{1, \dots, p\} \} \end{aligned}$$

4.6.3 Interaction Contracts

Twitter-like

The cardinality of the feature space is one, giving one type of interaction contract, $X_{1,1}$. This contract can be thought of as a state-synchronising datagram, allowing for a consistent representation of the *Twitter-like* 'blog' state.

WhatsApp-like

Again, the cardinality of the feature space is one, giving one type of interaction contract. Given that there are n different group chat instances, however, we can consider the interaction contract $X_{i,j}$ to denote a message exchange between group chats i and j . If $i = j$, this contract denotes a state synchronisation update, similar to the *Twitter-like*. Otherwise, this represents a 'forwarding' action from one group chat to another.

Facebook-like

To avoid enumerating all $\text{card}(\mathcal{C})^2 = 9$ types of interaction contract grammar (profile to group chat and vice versa, profile to shop and vice versa, etc), we choose a subset based on the *profile* feature. With this, we can form three kinds of contracts between (other) profiles (p), groupChats (g) and shops (s):

1. $C_p \xleftrightarrow{X_{p,p}} C_p$ – a contract between two profiles, used for friend requests
2. $C_p \xleftrightarrow{X_{p,g}} C_g$ – a contract between a profile and a groupChat, used for applications
3. $C_p \xleftrightarrow{X_{p,s}} C_s$ – a contract between a profile and a shops, used for transactions

4.6.4 Methodology

Having represented three example platforms in the form $\langle \mathcal{C}, \mathbf{X}, \mathcal{R}_{nf} \rangle$ by enumerating the feature domains and implementing them as bounded context, giving an example set of interaction contracts, and supplying a (common) set of example non-functional requirements, they are in a suitable form to be mapped onto a plugin architecture. Accordingly, for each platform we summarise the necessary feature variety and cardinality, as well as the subset of functional requirements and interaction contracts defined in the above discussion in Table 4.2.

Criteria↓ / Platform→	<i>Twitter-like</i>	<i>WhatsApp-like</i>	<i>Facebook-like</i>
Feature Variety	Homogeneous	Homogeneous	Heterogeneous
Feature Cardinality	One	One	Many
Functional Requirements	Group-chat	Group-chat	Group-chat; Profile; Shop
Interaction Contracts	Blog update	Chat update; Message forwarding	e.g., Friend request; Transactions; Group-chat
Architectural Paradigm	Plugin	Plugin	Plugin

Table 4.2: Comparative abstraction of *Twitter*-, *WhatsApp*-, and *Facebook-like* platforms across dimensions of feature space, functional requirements, and interaction contracts

Per Algorithm 1, by implementing each of the bounded contexts as their own plugin, and embedding the non-functional requirements into a kernel that has user registration and datagram storage, say, each of the plugins can be dynamically loaded to create a cohesive platform in the form of a plugin architecture.

4.7 Summary

In summary, this chapter has contributed one further step towards proving the existence of a viable alternative to the privately-owned monopolistic platforms currently dominating the digital social media landscape. Chapter 3 demonstrated that this alternative was *specifiable* by formalising three high-level components for conducting ethical platformisation based on a plugin architecture. However, just as the Turing Machine would be a curiosity if there were no *Universal* Turing Machine that can solve any problem solvable by an algorithm, such a plugin architecture would be irrelevant if it were a ‘one-trick pony’, i.e., something that only works in one domain. The platformiser, and in particular the platforms it produces, must be *computable*.

We therefore needed to prove *platform completeness* – that there is a plugin architecture that satisfies the requirements of any platform specified in terms of its bounded contexts, interaction contracts, and non-functional requirements (i.e., any platform). Since we have defined a constructive mapping (f) that maps an abstract representation of all *platforms* ($\mathcal{P}$) onto an abstract representation of all *plugin architectures* ($\mathcal{PA}$), this plugin architecture must be capable of implementing any platform, and is therefore platform complete. In more detail, the contributions of this chapter are to have:

1. Constructed a universal formalism for an arbitrary software platform $\mathcal{P}$ based on Evans’ domain-driven design [121] and “n-layer” web applications [124] in the form $\mathcal{P} = \langle \mathcal{C}, \mathbf{X}, \mathcal{R}_{nf} \rangle$
2. Constructed a universal formalism for a plugin-based platform architecture $\mathcal{PA}$ based on Baldwin and Woodard’s platform architecture; [2] in the form $\mathcal{PA} = \langle K, \mathcal{L}, \mathbf{X}_K \rangle$;
3. Specified an algorithmic, refinement-mapping function showing that any platform of the form $\langle \mathcal{C}, \mathbf{X}, \mathcal{R}_{nf} \rangle$ can be expressed in the form $\langle K, \mathcal{L}, \mathbf{X}_K \rangle$, i.e $\mathcal{P} \mapsto \mathcal{PA}$;
4. Used this mapping function to formalise a constructive refinement from an abstract platform architecture to a concrete plugin-based implementation; and
5. Applied this algorithm to given examples of *Twitter-like*, *WhatsApp-like* and *Facebook-like* platforms as plugin-based systems.

As a corollary, we concluded that this plugin architecture can therefore be used to implement any possible software platform, and is therefore *platform complete*.

In conclusion, we contend that the proof of platform completeness constitutes a novel intervention into the techno-politics of platformisation. The domination of digital social media by a few private and trans-national corporations may well be a product of network effects at the application layer of the Internet and a feature of client-server architectures (that blockchain-based architectures attempt to redress [69]) – but whatever else is claimed, *it is not inevitable*.

In some senses we have been here before. For example, in the domain of sustainable common-pool resource management, it was claimed that depletion (exhaustion) of the resource was inevitable – the so-called tragedy of the commons – and the solution was privatisation or coercion. However, Elinor Ostrom won the Nobel Prize for Economic Science in 2009, for her extensive fieldwork demonstrating that, throughout history and geography, communities have themselves managed to sustain common-pool resources over generations [58]. They achieved this by forming what she called *institutions* that exhibited eight specific features, one in particular being *self-determination*. (Interestingly, successful and sustained Minecraft hosts are customised with plugins which implement these features [143].) Similarly, Graeber and Wengrow [37] observe that the increased scales of alignment in social groups are associated with an increased need for hierarchy – but it is not inevitable that governance should get ‘stuck’ in any particular form of hierarchical social arrangements. The requirement here is community self-governance that allows for seamless transitions between different modes of decision-making according to the legitimate consent of the participants.

Privately-owned digital social media platforms are predominantly transactional, extractive and depletive. *It does not have to be this way*. Therefore, the potential social impact of our platform completeness result is to show that a *platform complete* plugin architecture can, *in principle*, satisfy arbitrary community-centric, stakeholder-driven requirements. We next demonstrate this *in practice* through the implementation of the *PlatformOcean* platformiser in Chapter 5, and its application in Chapter 6.

Platformiser Implementation

Contents

5.1 Introduction	81
5.2 Meta-Platform	83
5.3 Platform	86
5.3.1 Backend	86
5.3.2 Frontend	90
5.4 Plugins	97
5.4.1 Security: Remote Code Execution	101
5.5 Software Development Kit	103
5.6 Software Stack	104
5.7 Comparison to Existing Extensible Web Architectures	106
5.7.1 Nextcloud	107
5.7.2 Drupal	108
5.7.3 WordPress	109
5.8 Summary	109

5.1 Introduction

This chapter presents *PlatformOcean*, one possible implementation of a *platformiser*. This platformiser is available at <https://github.com/PlatformOcean>

The *PlatformOcean* platformiser comprises three components. A (vanilla) *platform*, provides standard distributed messaging functionality and a lifecycle mechanism for injecting *modules*. The modules manifest themselves as *plugins*, serving as remote pieces of self-contained, parallel-developed code that extend the platform. Finally, a *meta-platform* is used to instantiate both

vanilla and pre-customised platform instances, and serve as a repository for storing the uploaded plugins.

This high-level architecture is demonstrated through the different information pathways in Figure 5.1. Central to the *PlatformOcean* ecosystem is the meta-platform, which provides the mechanism for instantiating distinct platform instances (or *backends*, discussed in Section 5.3.1). These platform instances can subsequently be connected to with the provided client tool (or *frontends*, discussed in Section 5.3.2). These client tools receive the metadata for the set of plugins used to customise the platform (discussed in Sections 5.2 and 5.4) which can subsequently be resolved back to the meta-platform to fetch the plugin as a dynamically-injectable remote component, and instantiate it in the platform.

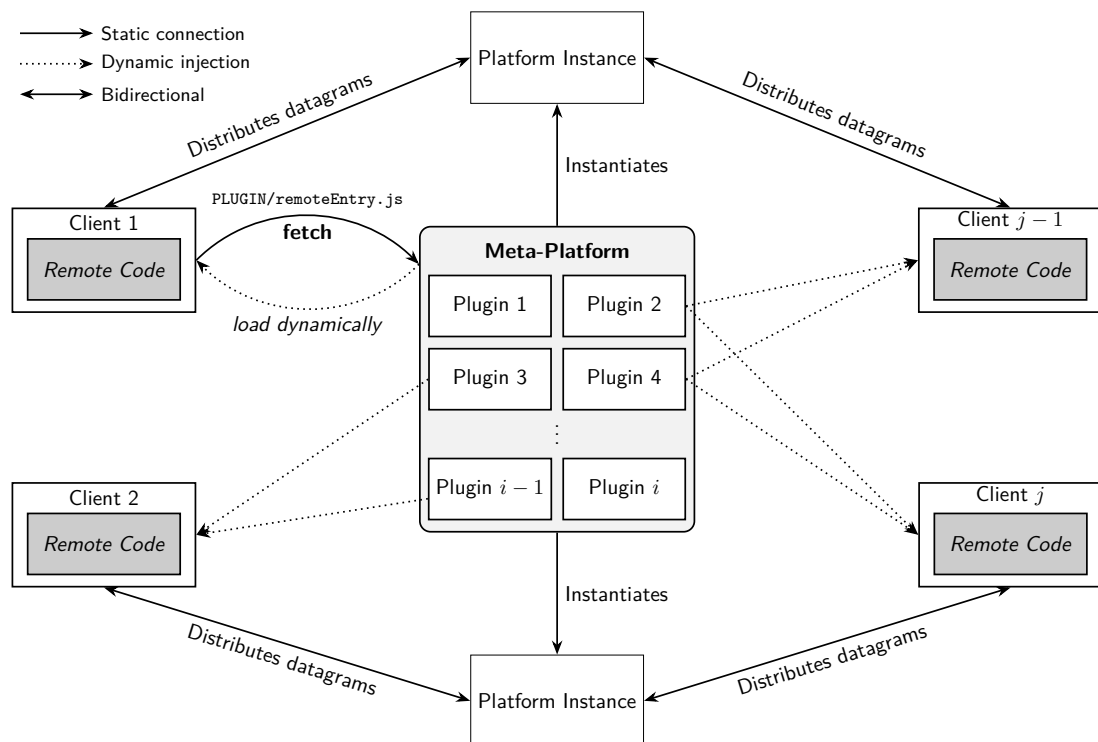


Figure 5.1: Concrete *PlatformOcean* system architecture. Multiple clients connect to multiple servers, and dynamically inject multiple plugins

Reifying this abstraction, this section decomposes the full-stack *PlatformOcean* architecture into three stages. Firstly, at the top level, Section 5.2 defines the *meta-platform* (i.e., a platform for platforms) which serves as a single, centralised point of contact for instantiating the *platform* instances. Secondly, Section 5.3 defines these platform instances in their ‘vanilla’ state, and how they provide the necessary tools for user registration, data storage, and instance discovery. Finally, in order to provide the bespoke platform functionality, Section 5.4 describes how these platforms can be customised with *plugins* – dynamically injected software components that provide a user

interface (UI) similar to a “micro-app” or “micro frontend” [144], allowing for users to interact via datagram messages sent across the platform.

5.2 Meta-Platform

The *meta-platform* provides a single point of access for both users and developers. For a developer, the meta-platform acts as plugin repository – i.e., an endpoint for plugin developers to submit their designs for subsequent download/inclusion in a platform instance. This section details the implementation of Component 1 from Section 3.2.

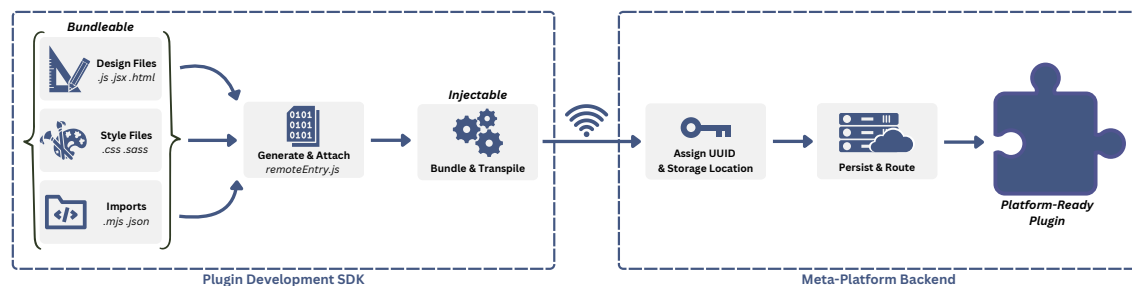


Figure 5.2: *Plugin-development pipeline*. Users leverage the SDK to compile design files, style files, and imports into a single **bundleable** object. The SDK then generates and attaches a *remote entry* file to the bundle, before transpiling it into dynamically-injectable *JSX* code. This **injectable** is then sent to the meta-platform backend, where it is unzipped, receives unique identification, and is persisted in storage

The **bundleable** is the key data structure underpinning *PlatformOcean*’s plugin representation. In order to provide a fully customisable, generic system architecture, developers must be able to codify plugins without restriction. Therefore, developers can leverage external libraries, various stylings and multiple design files to specify their plugin UI. The sole requirement is that the plugin, when finished, can be *bundled* (i.e., statically analysed and converted into one or more optimised output files) in order to be sent over a network and uploaded to the *meta-platform*. Hence, a bundleable is defined (recursively) as anything that can be *bundled*. In this context, a bundleable manifests itself as extending a set of multiple, individually bundleable components. The minimal set of requirements for the bundleable is: *design files* – the raw *JSX* code that codifies the functional state of the plugin; *style files* – the raw *CSS* code that defines the stylings of the plugin component (in an *HTML-compatible* format); and the *imports* – the set of external libraries used in specifying the plugin.

Having fully specified a plugin, a user bundles their bundleable into a platform-appropriate format by using the plugin development software development kit (SDK). This toolchain generates

a *remote entry* ‘meta’ file (i.e., a file describing the file structure) that can be remotely accessed, allowing the plugin to be injected into a platform instance. Once the *remote entry* file has been attached to the bundle, thereby creating an **injectable**, the injectable is *transpiled* into a uniform JSX representation, *minified* to reduce the size of the plugin bundle, and *hashed* to give the encoded plugin injectable a unique identifier (thereby allowing for multiple plugins of the same name). Figure 5.2 demonstrates this process.

The role of the *meta-platform*, then, is to run a protocol that accepts the uploaded plugin, persists it in a database, and exposes a route to the *remote entry* file. This requires three additional components to exist on the *meta-platform* server: a static endpoint for the upload route, a database schema for persistence, and a dynamic file store for hosting and exposing the address of the plugin’s entry point.

Parameter	Range	Meta-Platform	Platform
<i>pluginID</i>	<i>UUID</i>	✓	✓
<i>pluginLoc</i>	<i>URL</i>	✓	✓
<i>instanceID</i>	<i>UUID</i>	✗	✓
<i>versionID</i>	<i>semver</i>	✓	✓
<i>author</i>	<i>string</i>	✓	✗
<i>pluginName</i>	<i>string</i>	✓	✗
<i>dependencies</i>	[<i>UUID</i>]	✗	✓

Table 5.1: Plugin metadata schema for *meta-platform* and *platform* components

Having produced the plugin injectable, a static connection can be formed with the (backend of the) meta-platform, enabling the plugin to be uploaded as a *multipart file*. Figure 5.2 separates this functionality into two ‘phases’: an initial ‘bundling’ phase, which occurs through the plugin development SDK, and a subsequent ‘persistence’ phase which occurs on the meta-platform. These two phases are ‘bridged’ via the meta-platform’s API, facilitating a static connection between the two pieces of software.

The plugin bundle can then be unzipped (as the bundle must be sent across the internet as a serialisable zip file), written to the server’s local memory, and exposed as a static file using the paradigm of a static file server. The plugin’s *metadata* (i.e., data about the data) persists, which helps with organisation and identification, as well as minimising the footprint of the response when a client queries the database for the list of available plugins. For each plugin, the *meta-platform* stores the *pluginID* as a *UUID* for unique identification, as well as its (human-interpretable) name, its author, and the *URL* of the file location as *strings*. This schema, as well as the simplified version used for plugin representation in a platform instance, is shown in Table 5.1.

Upon redistribution to a platform, however, the persistence metadata becomes redundant.

Plugin composition is defined at the platform level, rather than the repository level. Accordingly, the injection metadata encodes which plugins are loaded and how they interact. An additional *instanceID* field is supplied to enable multiple copies of the same plugin, and a *dependencies* array is attached (on injection) to codify which other plugins are ‘listened to’, to enable rudimentary plugin-to-plugin federation (explored later in Section 5.4). The *pluginLoc* field becomes critical in facilitating the behaviour of a *federation* of meta-platforms (a CDN or P2P network, say), whereby the same plugin may be fetched from a variety of different repositories.

Using the plugin architecture formalism outlined in Equation 4.7, a platform is representable by a common *kernel*, K , and an array of dynamically loadable plugins, $\mathcal{L}$. By storing the plugin metadata as well as the transpiled plugin file, any pre-configured platform instance can be represented/encoded by a counter of *pluginTypes*, such that:

$$\mathcal{L} \equiv \mathbf{map}(\text{pluginType} \rightarrow \text{int})$$

where, upon instantiating a platform, this encoding can be used to retrieve the set of plugins and their corresponding count needed for generative replication.

This metadata schema implements many of the key abstractions from other artefact repositories. For example, the *Node Package Manager* (NPM) serves artefacts in the form of packages and libraries to applications built under *Node.js*: a free, open-source, cross-platform *JavaScript* runtime environment. Authors of a *Node*-compatible library supply a *package.json* file, which provides the necessary metadata for persistence in the repository. Much like the *PlatformOcean* implementation, NPM provides publication metadata in the form of *name*, *version*, and *keyword* fields, and dependency and execution metadata through the *dependencies* and *scripts* fields, respectively. An exhaustive list of fields can be found at <https://docs.npmjs.com/cli/v11/configuring-npm/package-json>¹.

Similarly, the *Python Packaging Index* (PyPI) acts as a repository of software for the Python programming language, where authors of Python libraries supply a *pyproject.toml* file to encompass the metadata for persistence. Again, the *name* and *version* fields (among others) codify the publication metadata, and the *dependencies* and *scripts* codify the dependency and execution metadata. An exhaustive list of fields can be found at <https://packaging.python.org/en/latest/guides/writing-pyproject-toml/>².

¹Last accessed 12/05/2026.

²Last accessed 12/05/2026.

In particular, the schema proposed in Table 5.1 encodes a core architectural distinction: the meta-platform stores canonical plugin identity and publication metadata, whereas the platform stores runtime instantiation metadata. This abstraction therefore differs from a ‘conventional’ artefact repository in order to synergise with the structure of the *PlatformOcean* component stack. The meta-platform is responsible for mechanisms relating to plugin storage and distribution, while the platform is responsible for mechanisms relating to plugin injection and runtime composition.

5.3 Platform

5

Functionally, the role of the (micro-)kernel is to analogous to a ‘coordinator’ agent. The kernel provides the external modules with generic, architecture-agnostic services such as scheduling, life-cycle management, and memory management [62]. Furthermore, in a network configuration, the kernel also provides an inter-process communication facility, allowing the independent modules to interact and exchange data [62]. This helps synchronise the global state between the modules, as the kernel formalises the mechanisms for module-to-module interactions. Per the principles of domain-driven design, this identifies the kernel as a *context map* [121], [128], [129].

In providing the mechanisms for inter-module communication, the kernel also plays the role of a message broker. To keep the kernel and modules decoupled, a key principle of the microkernel design paradigm, message passing is central to the structuration [62]. Accordingly, a generalised message-passing foundation is supplied that allows the unconstrained development of orthogonal modules without the need for a universally agreed-upon grammar [62].

The platform can reductively be thought of as a data storage and distribution system. The backend of the platform, when instantiated as a locally-hosted server, stores users, datagrams, and (a reference to) the plugins used to customise it. The frontend, however, yields a user interface (UI) for interacting with the backend, through a set of predefined protocols passed to the abstract plugin components. Accordingly, we describe the *backend* operation of the platform in Section 5.3.1, and its *frontend* in Section 5.3.2. This section details the implementation of Component 2 from Section 3.2.

5.3.1 Backend

To access a prospective platform, each platform serves as a ‘beacon’, by periodically emitting a known datagram to a predetermined multicast address over HTTP. Subsequently, upon hearing

this discovery message, a prospective client responds via a unicast message to the exposed platform address. This process is termed *HTTPU*. Together, this protocol stack implements the *Universal Plug and Play* (UPnP) service discovery protocol, serving to expose the addresses of all prospective platforms on the local network [145], [146].

Parameter	Range
<i>clientID</i>	<i>UUID</i>
<i>username</i>	<i>encryptedString</i>
<i>password</i>	<i>encryptedString</i>
<i>locked</i>	<i>boolean</i>
<i>role</i>	enum {Member, Owner, Banned}

Table 5.2: *userData* schema for persistence of members in a platform

Having selected an address, the user can then either retrieve their unique platform *clientID* if they're a member, or submit a request to join. Upon joining a platform instance, a given user is supplied with platform-specific *userData* that provides identification at the platform level. Users gain a unique identifier in the form of a *UUID*, supply a username and password that undergoes end-to-end encryption, and get assigned a *role* in the form of an *enum* denoting the membership state of the client: a platform *owner*, a 'standard' *member*, or *banned*, if the client has been blacklisted from a platform. The full schema for representing user data is shown in Table 5.2, demonstrating how platform membership is coordinated and unique identification is assigned. The full registration lifecycle, from platform detection to voluntary association, is further demonstrated in Algorithm 2.

Alternatively, depending on the intended hosting option of the platform instance, a second discovery protocol may be used. For example, while a local laptop or Raspberry Pi may be conducive to discovery via UPnP, hosting on a cloud requires a known internet address. Discovering previously-unknown IP addresses requires UPnP to return a set of prospective platform addresses on a local network. If the endpoint is known, the same *signOn* protocol from Algorithm 2 can be used directly, without the need for running a service discovery protocol beforehand.

In the spirit of ecosystem engineering, it is also possible to delegate a *PlatformOcean* platform (or indeed a *meta-platform* instance), operating at a known address, to serve as a kind of service registry. In this way, a single endpoint can return the list of publicly-exposed platforms running on the Internet.

When the platform is customised with plugins, it gains two further features. Firstly, the platform can be thought of as a 'coordinating agent' for all of the connected clients in the network. It stores the current list of plugins that have been added to the platform, as well as an *instanceID*

Algorithm 2 PlatformOcean Instance Joining Algorithm

```

1: define struct client:
2:   client.platformAddress  $\leftarrow$  null
3:   client.clientID  $\leftarrow$  uuid.null
4:   client.socketSession  $\leftarrow$  null

5: procedure GETUSERDATA(endpoint, name, pass)
6:   if not userExists(endpoint.names, endpoint.passes) then
7:     endpoint.register(name, pass)
8:   userData  $\leftarrow$  endpoint.lookup(name, pass)
9:   return userData
10: end procedure

11: procedure SIGNON(client, network)
12:   endpoints  $\leftarrow \subseteq$  UPnP(network)
13:   for all endpoint  $\in$  endpoints do
14:     client.platformAddress  $\leftarrow$  endpoint  $\triangleright$  bind platform address to client
15:     client.socketSession  $\leftarrow$  stomp.Over(endpoint)
16:     name, pass  $\leftarrow$  *user input*  $\triangleright$  process sign-on from client UI
17:     platformUserData  $\leftarrow$  getUserData(endpoint, name, pass)  $\triangleright$  From Table 5.2
18:     if platformUserData.role  $\neq$  role.Banned then
19:       client.clientID  $\leftarrow$  platformUserData.clientID
20:     end if
21:   end for
22: end procedure

```

key to uniquely identify each plugin instance. This is different to the *pluginID* key in the *meta-platform*'s plugin metadata, as the platform can host multiple copies/instances of a single type of plugins. Consequently, a single plugin identifier maps to multiple instantiations. This data structure is formalised in Table 5.1. Secondly, the platform stores all recorded plugin data (the datagrams specified by the plugin) and redistributes it across all of the connected clients via a websocket connection. This allows for real-time interactions over the platform's plugins. This data structure is formalised in Table 5.3. By implementing discovery mechanisms compatible with any prospective platform client, lifecycle mechanisms compatible with any prospective plugin, and brokerage mechanisms compatible with any prospective datagram, this backend implementation successfully implements the generic platform kernel, *K*, defined in Section 4.4.1.

This message distribution algorithm is contingent on two datatypes: the **request** (representing an inbound message from the client to server) and the **response** (representing an outbound message from server to client). Common to these datatypes is the *datagram* object in Table 5.3: all arbitrary datagrams must encode the provenance of the datagram through who it came from (its *senderID*) and where it came from (its *pluginID*), as well as the *payload* of the datagram itself. Given that the payload codifies the semantics of the plugin state and is decoupled from the platform backend, an added encryption layer can cleanly sit on top of the transport mechanism. Both plugins and

platform emit and receive arbitrary datagrams, so as long as there is a mutual agreement between the encryption and decryption mechanisms, these datagrams can become opaque to the platform owner.

Parameter	Range	Request	Response
<i>datagramID</i>	<i>UUID</i>	✗	✓
<i>senderID</i>	<i>UUID</i>	✓	✓
<i>pluginID</i>	<i>UUID</i>	✓	✓
<i>payload</i>	<i>JsonNode</i> $\supseteq$ Object	✓	✓
<i>protocol</i>	<i>enum</i> { <i>create</i> , <i>update</i> , <i>delete</i> }	✗	✓
<i>shouldPersist</i>	<i>bool</i>	✓	✗
<i>recipients</i>	[<i>UUID</i>]?	✓	✗

Table 5.3: Schema for metadata envelope to fully characterise arbitrary datagram in platform

A **request** from the client attaches a boolean *shouldPersist* field to the datagram, to inform the platform whether or not to commit the message to memory. Upon receiving this data, the backend strips this field and generates a **response** schema. The backend generates and attaches a unique *datagramID*, to identify the datagram, as well as the *protocol* with which the datagram was sent – i.e., *create*, *update*, or *delete*. This allows the frontend to conditionally handle the received **response**.

To handle targeted datagram routing, the **request** envelope also contains an (optional) *recipients* array. In order to avoid redundant networking operations (and offer increased message security), clients subscribe to a bespoke topic identifying provenance via the plugin’s *instanceID* and user’s *clientID*. The platform backend then iterates over the relevant clients in the recipients array and directs the datagram to the topic address. Null recipients arrays are interpreted as a broadcast message, and therefore exposed to all connected clients.

More generally, the platform implements a universal infrastructural wrapper where developers get the platform metadata for ‘free’ via a common envelope. This allows them to specify their own, bespoke payload to arbitrarily codify the plugin state. A similar approach is taken with *HTTP*, which standardises transport while payloads remain arbitrary, or *MQTT*, which standardises messaging while topics/payloads vary. The strength of this design paradigm is that plugins become coupled to the *interaction contract* defined at the plugin level, rather than to the platform implementation itself. Consequently, plugins become flexible, first-class components that interact through a loosely-coupled platform. This therefore reduces platform lock-in, as plugins can migrate and participate in *any* platform instance without necessitating platform-specific rewrites.

A similar separation-of-concerns paradigm is implemented by *Signal*, as the transport (or, indeed, *service*) metadata is decoupled from the application content. The *SignalService* envelope

[147] achieves this by exposing a *sourceUuid*, for provenance, alongside a *type*, for codifying the protocol, as well as *content*, which defines a family of possible message interaction mechanisms. Where the *PlatformOcean* paradigm diverges, however, is that it arguably generalises this process from more than just carrying a message to informing a state transition. *Signal*'s envelope is designed to support secure communication and cryptographic message delivery, whereas *PlatformOcean* utilises the envelope as a platform-level coordination mechanism. Again, the use of an arbitrary, plugin-defined payload ensures that the transport layer remains agnostic to domain-specific schemas.

5

5.3.2 Frontend

Having specified the representation of arbitrary datagrams, and the backend system that allows for their redistribution and storage, we now describe the operations available on the platform's frontend for creating and distributing these datagrams.

The frontend component of the platform is responsible for providing a user interface for interaction with its backend. In serving as an implementation of a decentralised, plugin architecture, the platform frontend is capable of dynamically connecting to multiple backends, and supporting a wide array of grammars specified on various plugins. This therefore allows a single frontend tool to connect to multiple different platform instances.

In order to support a variety of plugins – all of which will be designed with different grammars and functionalities – the frontend tool provides a *wrapper* component. This *plugin wrapper* provides a set of attributes and methods for interacting with an arbitrary platform instance, to comply with the protocol defined in Section 5.3.1. This list of frontend methods and attributes is shown in Table 5.4.

Internally, the wrapper component spawns a thread for connecting to a *WebSocket* route on a predefined platform instance. Through subscribing to this topic, all datagrams passed over instances of this plugin (identifiable by its *pluginKey*) are deserialised and stored within the *messageHistory*, giving a linear history of all persisted messages sent across the plugin. The platform backend operates as a kind of coordinating 'agent', by storing the 'true' history of datagrams from all connected clients. Synchronising this state across the system can therefore be potentially expensive, as the entire linear history must be distributed as a serialised array that grows proportionally to the number of stored messages. The plugin lifecycle therefore offers two mechanisms for state synchronisation. Upon successful injection of a plugin, the entire linear history is fetched

Attributes	
Name	Range
<i>client</i>	STOMP.js Client
<i>clientID</i>	UUID
<i>pluginKey</i>	UUID
<i>messageQueueLen</i>	int
<i>messageHistory</i>	[JsonNode]
Methods	
Name	Return Value
<i>pollMessageQueue()</i>	JsonNode?
<i>sendCreateMessage(JsonNode, bool)</i>	bool
<i>sendUpdateMessage(JsonNode, UUID)</i>	bool
<i>sendDeleteMessage(UUID)</i>	bool

Table 5.4: Attributes and methods provided by wrapper component

once, guaranteeing a consistent replication of the ‘true’ state defined by the authoritative backend log. Subsequent messages use a local history, where each redistributed message gets appended to the initial fetch. This ensures that each datagram is transferred exactly once following the initial synchronisation phase, rather than for every successive exchange, reducing the cumulative synchronisation overhead from $O(n^2)$ to $O(n)$.

Furthermore, high-frequency message passing is handled by a message queue implementation: a *FIFO* buffer stores unprocessed messages in a queue and exposes the queue length (see Table 5.4). Plugins can then (idiomatically) attach an event listener to watch the state of the queue (via a *useEffect*, say), allowing for efficient polling of unprocessed states. This allows plugin designers to conditionally render the datagrams to their UI, based on the grammar they have defined. These two methods can be considered as **read** operations.

The datagram transport layer is therefore composed of two phases. Datagram routing (and provenance) requires knowledge of which plugin and which client(s) should be consuming the message. Accordingly, on plugin injection, clients subscribe to a *WebSocket* topic characterised by the *pluginKey* and *clientID* from Table 5.4. This creates a distinct pathway for low-latency, bidirectional message-passing under the publisher/subscriber (pub/sub) paradigm. For correctness, the platform backend operates as the authoritative source of plugin state which the connected clients use to define the interface.

Furthermore, for handling high-frequency datagram exchange (for rapid state-synchronisation events, say), the platform client also implements a *FIFO* buffer. This allows incoming datagrams to be synchronously appended to a single, ordered queue, whilst plugin state updates are processed asynchronously. This design draws inspiration from message-oriented middleware such as *RabbitMQ*, where message production and consumption are decoupled, specifically through asyn-

chronous queue-based delivery [148].

PlatformOcean also adopts aspects of the *event sourcing* architectural pattern: plugins raise a state-mutating event through the payload, the platform backend persists these changes, and the plugin can rebuild its state by replaying its past stream of events [149, p. 113]. However, unlike a pure event sourcing system, the platform does not enforce a strictly immutable append-only event log, as state transitions may be modified through update operations or removed through delete operations *at the server level*. Nevertheless, this gives the benefits of minimising per-message overheads (as discussed earlier); the capacity for full state rebuilds, since plugins can simply retrieve the authoritative log; and event replay: the current state can be independently verified by any plugin by reading the past N logs [150].

The wrapper provides the three necessary methods for **CRUD** operations on the platform backend – i.e., a *create* method, taking as argument an arbitrary, serialisable datagram; an *update* method, taking as arguments the ID of the datagram to change and the datagram to replace it with; and a *delete* method, taking as argument the ID of the datagram to delete. These three methods can be considered as **write** operations.

In particular, the *create* method exposes a second parameter, defining whether or not the datagram should persist in the platform backend. The transport layer therefore supports both persistent and *ephemeral* datagram semantics. Persistent datagrams are stored as part of the backend’s authoritative log, whereas ephemeral datagrams are redistributed transiently without persistent storage.

The specification *and* interpretation of the datagrams’ grammar/semantics is performed entirely within the plugin context. While the platform envelope in Table 5.3 codifies the metadata for the transport layer, the structure and *meaning* of each interaction contract comes specifically from the arbitrary *payload* schema specified at design-time. Using this schema, plugin developers can perform *client-side demultiplexing* through the inclusion of a *contractType* field, say, similarly to *Redis Pub/Sub* [151] (exemplified in [152]). Consequently, messages are demultiplexed and interpreted dynamically at run-time according to the plugin’s internal payload grammar and implementation of the underlying *state machine* (See Section 5.4). This ensures that the application-level semantics are decoupled from the transport-layer delivery, which in turn *further* decouples the platform from its plugins. One example of an ‘idiomatic’ plugin implementation is shown in Listing 5.1, combining datagram metadata from Table 5.3 with the wrapper methods from Table 5.4.

```

1  useEffect(() => {
2    if (messageQueueLen === 0) return;
3
4    // extract full message envelope
5    const { message } = props.pollMessageQueue();
6
7    const handlers = {
8      contract_one: (msg) => {
9        /* handle contract_one schema using msg.payload */
10      },
11
12      contract_two: (msg) => {
13        /* handle contract_two schema using msg.payload */
14      }
15    };
16
17    const contractType = message.payload.contractType;
18    const handler = handlers[contractType];
19
20    if (!handler) {
21      console.warn(`Unknown contract type (${contractType}). Ignoring.`);
22      return;
23    }
24
25    handler(message);
26
27  }, [messageQueueLen]); // listen to changes in length of message queue

```

Listing 5.1: Client-side demultiplexing of datagrams in the *PlatformOcean* paradigm

The *PlatformOcean* ecosystem comprises a wide array of both plugins and instantiated platforms. When customising these platforms, plugins may be instantiated in one of two ways. A plugin may operate as a ‘stand-alone’ unit of functionality, existing independently and maintaining shared state across multiple connected clients. One example is a group chat plugin, in which the plugin – working in tandem with the platform – coordinates messages from various users. Alternatively, plugins may extend pre-existing plugins. In this case, they are explicitly dependent on other plugins and augment their behaviour by providing additional functionality. One such example is a translator plugin that can be used to extend the group chat plugin, mentioned above. This plugin can ‘intercept’ (i.e., listen to the topic of) the messages disseminated over the group chat plugin and translate them to a given language in its own plugin environment. Accordingly, the wrapper component in the frontend client tool defines two message-distribution protocols:

1. **Intra-plugin protocol:** This protocol disseminates datagrams across instances of the same plugin type on multiple clients (e.g., from a group chat on client x to the same group chat plugin on client y). Synchronising the local state of a plugin is independent of external dependencies, so plugins can subscribe directly to the appropriate websocket address on the platform without performing dependency checks. Messages are propagated using **write**

operations and subsequently consumed via **read** operations, using the full range of methods available in Table 5.4.

2. **Inter-plugin protocol:** This protocol allows a plugin to extend or react to a *base* plugin of a different type (e.g., a translator plugin listening to a group chat plugin). Rather than synchronising its own state, the extending plugin subscribes to the same topic as the *base* plugin to intercept messages. Given a potential chain of dependencies, Algorithm 3 is used to evaluate if the plugin is ready to be injected into the DOM. Moreover, because the *base* plugin already manages the state, the extending plugin operates purely as a listener, exposing only the **read** operations from Table 5.4.

A given plugin therefore has two data streams: the local state is synchronised using the *intra-plugin* protocol, and this functionality, and that of other plugins can be extended using the *inter-plugin* protocol, reflecting the dual roles of synchronisation and extension. Per the formalisation of interaction contracts in Chapter 4, the inter-plugin protocol implements the ‘general’ form of contract $C_{i,j}$, $i \neq j$, while the intra-plugin protocol implements the special subset $C_{i,i}$.

Algorithm 3 *PlatformOcean* Plugin Readiness Algorithm

```

1: define readyState  $\in$  enum{WAITING, LOADING, LOADED}

2: define struct Plugin:                                 $\triangleright$  Simplified representation for injection
3:   pluginKey  $\leftarrow$  UUID                                 $\triangleright$  Unique plugin identifier
4:   dependencies  $\leftarrow$  set[pluginKey]                 $\triangleright$  Set of dependencies for inter-plugin communication
5:   readyState  $\leftarrow$  WAITING

6: define PluginRegistry := map(pluginKey  $\rightarrow$  Plugin)     $\triangleright$  Global registry for plugin lookup

7: procedure CANINJECTPLUGIN(pluginKey)
8:   plugin  $\leftarrow$  registry[pluginKey]
9:   if plugin.readyState = LOADED then
10:    return true
11:  else if plugin.readyState = LOADING then
12:    return false                                 $\triangleright$  Or throw CycleDetectedError
13:  end if
14:  plugin.readyState  $\leftarrow$  LOADING
15:  for all dep  $\in$  plugin.dependencies do
16:    if not CANINJECTPLUGIN(dep) then
17:      return false
18:    end if
19:  end for
20:  plugin.isReady  $\leftarrow$  LOADED
21:  return true
22: end procedure

```

By locally federating plugins, and allowing interaction through the inter-plugin protocol, the system implicitly specifies a graph where each node represents a plugin, and each directed edge

represents a dependency. Resolving these dependencies is therefore equivalent to linearising a directed graph, such that all vertices are ordered. Accordingly, Algorithm 3 implements a modified *topological sort* in the *React* paradigm, where each plugin maintains a reference to the global registry and queries its readiness state based on the readiness of its dependencies.

A similar approach is implemented with the *Open Service Gateway Initiative* (OSGi) framework, which develops *bundles* in the form of modular, deployable, Java files [153]. Moreover, the OSGi *BundleContext* API acts as the primary interface for these bundles to interact with the OSGi framework. This interface exposes a mechanism in which bundles transition through a lifecycle state machine (INSTALLED, RESOLVED, ACTIVE) according to dependency constraints and service availability.

Accordingly, to comply with the *React* paradigm, this set of methods is passed into the (dynamically injected) plugin component. This allows for the plugin developer, at design time, to use the pre-defined set of platform methods. This hides the inner functionality of the platform and simplifies the development process. We define this process as *occlusion*, and describe it with greater detail in Section 5.4. Having defined a plugin's representation in the *meta-platform*, and appropriate methods and protocols ensured by the wrapper component, the full plugin lifecycle is described below:

1. **Plugin retrieval** – Given a *pluginKey*, the platform first consults the *pluginCache* for previously loaded plugins. If a cached version is unavailable, the plugin module is dynamically fetched from the *meta-platform*
2. **Sandbox generation** – A dedicated sandbox environment is generated in the form of an iframe. This isolates the plugin from the host platform by treating the markup as being from a unique origin (thereby preventing direct access to internal state or platform APIs), and establishes a safe boundary for remote code execution by blocking features that trigger automatically [154].
3. **Wrapper generation** – The wrapper prepares the *intra-* and *inter-plugin* protocols by evaluating the global dependency graph for a plugin. When a plugin is ready to be mounted, the relevant subscriptions are made to the hosting platform's websocket addresses.
4. **Injection and activation** – Once the sandbox is ready and all inter-plugin dependencies are fulfilled, the wrapper component is attached to the dynamically-fetched plugin component and rendered into the sandbox in the DOM.

The overarching design philosophy behind this middleware is to minimise the semantic assumptions imposed by the platform itself. Naturally, this creates a fine line between what is ‘technically achievable’ in a plugin context, and what starts to verge on ‘hacky’. Tension emerges between promoting functionality to a server level, thereby risking increased coupling between the platform and its plugins, and delegating functionality at the interaction contract level, thereby potentially sacrificing technical optimality. For example, timed message expiration could reasonably be implemented both ways. At the interaction contract level, plugins can simply attach a *timeToLive* field to the *payload* and manually invoke a *setTimeout* to call the *sendDeleteMessage* method. Alternatively, this mechanism could be promoted to the server level, whereby *timeToLive* becomes a necessary parameter to the *sendCreateMessage* API, and is handled server-side by the platform backend runtime.

Consequently, the current architecture adopts a deliberately conservative approach. In this implementation, the backend provides only a minimal set of transport, persistence, and synchronisation primitives through the metadata envelope in Table 5.3, and wrapper API in Table 5.1. This allows for bespoke semantics to be delegated to the plugin runtime and plugin lifecycle updates to be delegated to its associated state machine implementation. This therefore permits developers to exert full control over the plugin’s schema and lifecycle semantics *entirely* at the application layer instead of (perhaps unnecessarily) having to comply with an API that becomes entrenched at the platform level. To cover both bases, this approach can further be future-proofed through co-operation and co-design with the platformiser stakeholders. The platform interface can, and *should* evolve emergently through co-production with the community.

This aligns with the previous discussion in Section 3.6.2 concerning the necessary assumptions about trust in the socio-political substrate; beyond technical guarantees, the usability of the platformiser is also contingent on the responsibility of the ecosystem’s stakeholders. Similarly to the *Linux* kernel development model, new features are not introduced unilaterally, but instead *emerge* through iterative proposals and reviews from the community itself. This results in feature development becoming an evolutionary process, in which commonly recurring patterns are gradually elevated into production-level features, rather than being fixed *a priori* by the platform designer.

5.4 Plugins

Underpinning a bespoke, generative platform is its *plugins*. In the context of *PlatformOcean*, we leverage the *React* framework to produce a dynamic, responsive web app; as such, the plugins must be compatible with this framework too. To achieve this, we model the plugins as a *microfrontend* (MFE) – i.e., decomposing the monolithic platform frontend into multiple smaller, self-contained frontends that can be dynamically injected and developed separately [144]. This section details the implementation of Component 3 – manifesting *modules* as *plugins* – from Section 3.2.

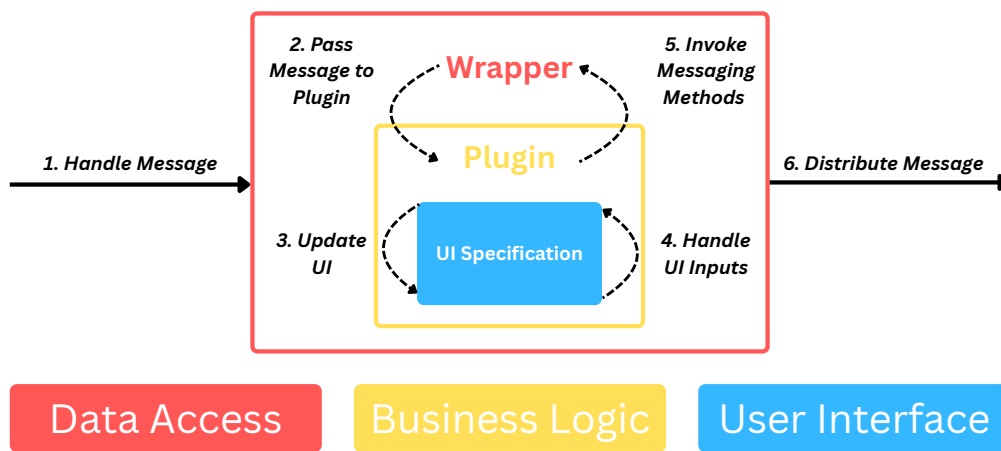


Figure 5.3: Plugin data-handling lifecycle in the context of an “*n*-layer” web-application architecture [124]

To maximise the expressiveness of plugins in order to ensure that users can develop applications unrestricted, we abstract their implementation. As MFEs, plugins must break down large, complex web applications into smaller, independent, and manageable frontend modules. A traditional *n*-tier layered architecture for web applications exposes three components. The *presentation tier*, or *interface layer* presents information and receives input data from users [155]. This data is then cached, or persisted, in the *data tier*, or *data access layer*, whereby it exposes methods for data access to ensure data integrity [155]. Finally, the *middle tier* or *business logic layer* defines the mechanisms for operating upon the data to update the user interface, thereby connecting the other tiers together [124], [155]. Given that the hosting platform is designed in React, plugin compatibility is guaranteed by modelling them as React *micro-apps*. These micro-apps characterise the MFE framework and are implemented as self-contained frontend units that plug into a larger host app. These micro-apps are small, independently developed and remotely deployed applications. By modelling them in such a way, each plugin can itself be a stand-alone React component and, using this React paradigm, accept arguments (or functions) into the component called *props*.

PlatformOcean therefore uses the analogy of classes to represent plugins. These classes ‘extend’ the wrapper component described in Section 5.3.2, in order to gain access to, and invoke, its methods. The method for rendering the user interface can then be ‘overridden’ in order to specify bespoke plugins. Furthermore, the plugin ‘subclasses’ all implement a common specification and contract with the wrapper ‘superclass’, allowing them to be dynamically added, removed, and swapped-out at runtime. A visualisation of this process is shown in Figure 5.3, comprising six steps:

5

1. *Handle Message* – Having subscribed to the topic defined on the relevant platform instance, the wrapper can deserialise any incoming messages into the standardised representation in Table 5.3
2. *Pass Message to Plugin* – Using React’s *props*, the wrapper passes the state down to the plugin, triggering a synchronous re-render
3. *Update UI* – Having triggered the re-render, the plugin’s UI updates based on the specified business logic
4. *Handle UI Inputs* – Following a re-render, the UI can be used to process user interactions with the plugin’s components
5. *Invoke Messaging Methods* – To synchronise the state across various client instances, the wrapper’s datagram dissemination methods can be invoked by the plugin
6. *Distribute Message* – Having codified the plugin’s state into a universal *JsonNode* representation, the wrapper serialises the datagram and distributes it over the network to the connected platform for the process to repeat

Plugins can be therefore be further abstracted and understood as mapping functions: they accept some input data, translate it into a corresponding change in their user interface, allow the user to interact with that interface, and then output an updated version of the data derived from the new UI state. In representing each plugin as an MFE, this state is best represented as a serialisable, universally expressive JSON object (i.e., a *JsonNode*), enabling unrestricted codification and distribution of plugin states across components. In this way, the system simply codifies a piece of state, feeds it into a plugin to drive its UI, and then receives the transformed state back. This creates a simple yet expressive pipeline for implementing UI-driven state evolution across multiple connected clients. Reductively, then, we can codify the required functionality of a plugin as follows:

$$f(inputData) \mapsto interfaceState$$

$$g(interfaceState) \mapsto outputData$$

The specific implementation of this mapping function gains granularity at the contract level. Listing 5.1 demonstrates how, by codifying message types into contracts, an otherwise agnostic payload can gain plugin-level semantics. This enables plugins to define multiple message types, each associated with specialised handlers and corresponding state transitions. In particular, the implicit codification of plugin behaviours at the contract level creates an interesting parallel with more conventional, object-oriented plugin implementations. In this paradigm, a module defines an interface which acts as a ‘contract’ for the functionality that a plugin must fulfil. The plugin can then be codified as a class which implements these interface methods. Symmetrically, *PlatformOcean*’s protocol/contract-oriented plugins becomes akin to a type signature, where instead of invoking a method, a datagram is demultiplexed based on some type. Furthermore, rather than passing in arguments to this method, *PlatformOcean* plugins can supply supporting metadata that define the same of these arguments in the form of a payload. Listings 5.2 and 5.3 show this symmetry.

```
interface Plugin {
    ReturnType operationA(InputTypeA input);
    ReturnType operationB(InputTypeB input);
    ...
    ReturnType operationN(ParameterSet
        parameters);
}
```

Listing 5.2: Object-oriented interface contract

```
Plugin.Contract = [
  {
    "type": "OPERATION_A",
    "schema": {
      "input": "InputTypeA"
    }
  },
  {
    "type": "OPERATION_B",
    "schema": {
      "input": "InputTypeB"
    }
  },
  ...
  {
    "type": "OPERATION_N",
    "schema": {
      "arg1": "ArgType1"
      "arg2": "ArgType2"
      ...
    }
  }
]
```

Listing 5.3: Protocol-oriented contract

Central to this plugin architecture is the dynamic remote component framework from *Module Federation* [156]. This framework allows for the dynamic inclusion of *React* components from a foreign endpoint at runtime, when only the URL of the hosting platform is known. *Module Federation* allows for pre-compiled *React* components to expose a *remote entry* file, which can be

injected into a host app (via a script tag) to register a component-providing module to the DOM's (document object model's) window object. From here, the component can be retrieved according to a pre-defined name and scope (declared during the bundling process).

Modelling and reinterpreting a plugin as an abstract, federated mapping function or class offers a series of advantages. Firstly, plugins implemented in this paradigm are *generic*. Developers can leverage the ubiquitous language of the *JsonNode* to achieve a universal state representation. This state can subsequently be read and mutated with the relevant mapping function, allowing for the specification of plugins to be potentially unbounded. Furthermore, *Module Federation* allows external libraries to be bundled into the final plugin component, meaning that developers can use them even if the relevant packages don't exist on the hosting platform.

Furthermore, the class-like structure of the wrapper component allows plugins to be *occluded*. The internal mechanisms of the platform can be hidden from view of the developer, as all networking and lifecycle methods are implemented by the open-source provider. The implemented plugin can simply extend the wrapper 'superclass', gain access to the distribution methods via *React props*, and override the render method.

The vanilla platform offered by the open-source provider, provides an array of plugin lifecycle methods. Using this software enables plugins to be *sandboxed* [20]. To support a wide array of customisation, the platform must also support multiple different plugins without unwanted crosstalk or corruption of the hosting software. As opposed to directly injecting the plugin into the main DOM where it may interact with other elements, the platform dynamically generates an *iframe* to provide an isolated environment with which the plugin can execute the remote code.

Whereas the injection phase of the lifecycle provides a mechanism for *sandboxing* the plugin, the removal phase allows plugins to be *hot-plugged* [20]. By leveraging *Module Federation*, plugins can be stored as pre-compiled JSX code. This code can then be fetched remotely from the *meta-platform* plugin repository and injected into the DOM. This allows plugins to be dynamically added to, and removed from, a platform at runtime, without a need for shutdowns or restarts.

Plugins are deliberately implemented to be decoupled from the implementation of the platform. Accordingly, the platform only needs to expose a common API wrapper that encapsulates each plugin within the universal coordination envelope defined in Table 5.3. Plugins are then injected within the platform wrapper in Table 5.4, which provides access to distributed, datagram-based communication primitives, while remaining agnostic to the structure of the transmitted data. As a result, plugin developers are free to define arbitrary, serialisable JSON schemas to codify the

application state. This is then coordinated by the platform backend which operates as both a canonical source of truth and message broker. The corresponding UI logic is similarly flexible, leveraging only *React* primitives. This enables plugins to function as self-contained, contract-driven modules within a shared runtime that is, critically, operable within *any PlatformOcean* context, irrespective of the plugin configuration. In summary, then, plugins are:

- *generic* – plugin developers can arbitrarily specify plugins using a universal state representation and external libraries;
- *occluded* – plugin developers can leverage the methods provided by the platform’s wrapper component to simplify the design process and remove the need for knowledge of the platform internals;
- *sandboxed* – plugins are injected into an isolated environment to prevent unwanted crosstalk and mitigate malicious behaviours;
- *hot-plugged* – plugins can be dynamically injected and removed from a platform at runtime; and
- *platform agnostic* – plugins are encoded as dynamically-injectable remote components, making them compatible with any *React*-based client.

5.4.1 Security: Remote Code Execution

The implementation of the *PlatformOcean* platformiser inverts the conventional component hierarchy. Rather than treating the platform kernel as the primary unit and plugins as conceptually peripheral [2], plugins are elevated to *first-class components*, with a minimal core retained solely as a vehicle for their composition and execution.

Following this approach, plugins necessarily maximise flexibility and expressiveness in order to implement the entire functional space of a platform (i.e., any bounded context per Definition 4 and Lemma 1). To avoid forced compliance with any given protocol, plugins are effectively unrestricted: they can define arbitrary state representations and implement corresponding state-transforming behaviours, subject only to the constraints imposed by serialisation and transport.

Consequently, this creates a trade-off with security. Unbridled execution environments grant malicious actors the possibility to perform *remote code execution* (RCE), wherein arbitrary, possibly

Byzantine code is executed within the plugin runtime context or underlying host system as a server-side script [157].

In total, *PlatformOcean* exposes three execution layers that may introduce potential vectors for RCE: state-mutating operations within plugins, interaction contracts defining the protocol interface between plugin and kernel, and the deserialisation and routing of datagrams within the platform kernel. For each of these vectors, we define a corresponding immunisation mechanism:

- *Plugin-level* – Plugins are *sandboxed* within *iframes*, ensuring strong isolation of execution contexts. This prevents plugins from directly accessing kernel memory, DOM state outside their scope, or other plugin instances, thereby limiting malicious code execution to the plugin boundary.
- *Contract-level* – Interaction contracts prevent “function-as-payload” semantics by strictly separating data from executable. All inter-component communication is restricted to serialisable datagrams, which are interpreted solely as raw JSON text. This reduces the possibility of control-flow injection through malicious messages.
- *Kernel-level* – The kernel enforces strict routing pathways for all incoming and outgoing datagrams, ensuring that deserialisation, validation, and redistribution occur in a controlled pipeline. While deserialisation *is* performed server-side, it is interpreted as raw text and persisted, rather than as an executable function.

Despite this, the internal plugin logic remains occluded from the kernel and obfuscated through bundling and minification. While this may not impact other plugins, or the operation of the system as a whole, malicious plugins may independently perform malicious actions. For example, in GPU-accelerated AI and ML systems, RCE attacks leveraging deserialisation vulnerabilities have been used to remotely deploy crypto miners [158]. A similar analogy exists with mobile apps; the application interface acts as an abstraction boundary that conceals the underlying implementation, thereby concealing malicious operation from the user. This vulnerability has manifested in a number of malicious applications, such as fake/cloned apps, illicit apps, Potentially Harmful Apps (PHAs), and malware [159].

Therefore, the issue of security cannot solely be addressed technologically. An underlying socio-technical framework is required. A promising option is a kind of ‘economy of esteem’, whereby users can rate and comment on plugins to either promote or discourage their usage. This is furthermore aligned at a ‘meta-level’ with the design values of *PlatformOcean*: the community can

```
Usage: oceanbundle -[b|e|p|s]

Edit a plugin in a local developer environment, and publish it to the
meta-platform

Options:
  --version    Show version number [boolean]
  -b, --bundle Bundle plugin to minified Bundle.zip [boolean] [default: false]
  -e, --edit   Host and edit plugin in local developer environment [boolean] [default: false]
  -p, --publish Publish plugin to repository [boolean] [default: false]
  -s, --scaffold instantiate blank project for plugin development [boolean] [default: false]
  --help      Show help [boolean]

You must provide at least one (valid) option.
```

Figure 5.4: Help screen for the *OceanBundle* SDK, enumerating the various options for developing a *PlatformOcean*-compatible plugin

self-organise using the meta-platform as a substrate to curate the visibility and reputation of the plugins it provides. This aligns with the philosophy of parallel trust streams and the requirements for socio-political trust introduced in Section 3.6.3.

5.5 Software Development Kit

Figure 2.2 illustrates that the platformiser ecosystem can be decomposed into two further interoperable ecosystems. The *artefact* ecosystem comprises the ocean of third-party modules and platform distributables, which is underpinned and maintained by the *user* ecosystem, comprising platform instantiators, plugin developers, and users that fulfil both roles.

In order for third parties to integrate their software with platform infrastructures, a relevant *software development kit* (SDK) is essential [13], [53], [160]. An SDK has no standardised form, but provides the necessary tools and toolchains for third-parties to develop add-ons or plugins that are compatible with a particular hosting platform. This section therefore introduces *OceanBundle*, the *PlatformOcean* SDK enabling third-parties to produce plugins that are compatible with any *platform* in the *PlatformOcean* ecosystem.

OceanBundle is a *command-line interface* (CLI) tool, that provides a suite of functionality to streamline the plugin development process. Figure 5.4 illustrates the landing screen for the SDK, generated through the `-h` command-line argument.

In total, the *OceanBundle* SDK provides four pieces of functionality:

- **Scaffold** – invoking the `-s` command-line argument populates a given folder with the necessary skeleton for a properly structured plugin. This generates a preconfigured *manifest* file,

creating the namespacing and metadata for remote plugin injection; *webpack* scripts for a local development environment; and *webpack* scripts for a production environment.

- **Edit** – invoking the `-e` command-line argument spawns a browser instance to host the plugin in local scope. In order to replicate the distributed networking functionality, where plugins send datagrams over the network between clients, the code generated from the **scaffold** command provides a mocked up version between two ‘instances’ using a parent component to simulate a server.
- **Bundle** – invoking the `-b` command-line argument minifies the plugin, reducing its file size, and attaches a *remoteEntry* file to facilitate dynamic inclusion of the plugin in a *PlatformOcean* platform.
- **Publish** – invoking the `-p` command-line argument uses the bundle generated from the **bundle** command and sends it as a multi-part file to the meta-platform, thereby completing the plugin development process.

5.6 Software Stack

The *PlatformOcean* platformiser consists of multiple ‘moving parts’ spread across various technical artefacts. For each of these artefacts – the *meta-platform*, *platform*, *plugins*, and *SDK* – a different set of functionalities are performed, all requiring the use of a specific software technology. We outline these technologies in Table 5.5.

Function	Technology
Meta-Platform	
Server (Brokerage)	Java w/ Spring-Boot
Server (Persistence)	MySQL
Platform	
Server (Brokerage)	Java w/ Spring-Boot
Server (Persistence)	H2 Embedded
Client (Interface)	JavaScript w/ React
Client (WebSocket)	StompJS
Client (Plugin Injection)	JavaScript w/ Module Federation
Client (App)	JavaScript w/ Electron
Plugins	
Interface Specification	JavaScript w/ React
Bundling	JavaScript w/ Webpack
Injectable Generation	JavaScript w/ Module Federation
SDK	
CLI	JavaScript w/ Node

Table 5.5: Technologies used to implement key functionalities in the *PlatformOcean* platformiser

The meta-platform implemented in Section 5.2 comprises two core functions, serving as both a message broker/controller/endpoint, and a persistence database. The entire meta-platform is deployable as a *maven* executable that uses *Java* with the *Spring Boot* framework to codify the controller, service, and repository subsystems. The language and framework provide mature concurrency primitives, strong networking support, and *Java Virtual Machine* (JVM) portability to enable multiple hosting options (See Figure 3.3). Persistence is implemented using MySQL due to its scalability and security, and the open source repository aligns with the *PlatformOcean* design value of *serving public interest*, in Section 3.2.

The platform implemented in Section 5.3 comprises two core sub-components: a server-driven backend that offers message brokerage and persistence, and a client-driven frontend that enables users to interface with any backend platform instance. Server-side, a similar stack to the meta-platform is used: *Java* with *Spring Boot* is again favoured for the concurrency support and an embedded server model that simplifies the development of platform instances. Furthermore, *Spring Boot* provides strong observability features, such as health checks (heartbeat pulses) and debugging for datagram identification. Rather than MySQL, however, H2 is preferred since it serves as an *embedded* database, thereby allowing platform instances to work *out of the box* (OOB), and lower the barrier to entry such that ‘anyone can own a platform’. Prioritising usability and lack of friction over the production-grade MySQL is a therefore a necessary trade-off for ensuring the democratisation of platform ownership.

The client-driven frontend of the platform provides a user interface for registration and platform interaction, a mechanism for real-time datagram transfer over WebSockets, support for plugin injection, and a means of packaging these capabilities into a stand-alone application. The client tool is primarily implemented in *React*, as it enables the rapid construction of dynamic and responsive user interfaces with a low barrier to entry. This design choice is motivated not only by platform implementation considerations, but also by plugin developer accessibility, as it lowers the barrier to entry for frontend programming and thereby supports the democratisation of plugin development within the ecosystem. Datagram coordination is handled using the *StompJS* library, as this is a web-native, browser-first messaging abstraction that best aligns with *PlatformOcean*’s deployment model, rather than a general IoT broker protocol using MQTT or AMQP, say. Plugin injection leverages a slightly modified ³ *Module Federation* implementation for fetching dynamic remote modules. Finally, while the client tool *can* be operated in the browser (and shared as an application using a minified build folder), we also supply a stand-alone desktop application that allows for

³The *promisified* remote module is resolved directly to default rather than fed into a *React.lazy* loader, giving greater flexibility over lifecycle management and error boundaries

OOB usage. This is generated using *Electron.js*, a framework for building desktop applications by embedding *Chromium* and *Node.js* into the *Electron* binary, due to its strong synergy with *React* and ease of use.

All plugins of the kind described in Section 5.4 are codified as *React* MFE artefacts. Accordingly, all plugin interfaces are programmed using the *React JSX* specification language, combined with standard *JavaScript* for invoking the *PlatformOcean* wrapper methods. The plugin specification language is inextricably tied to the platform implementation – a platform written in *React* must, and can only, inject *React* plugins. As aforementioned, *React* allows for the implementation of mature, dynamic, and responsive browser-based artefacts with a low barrier to entry, making it ideal for democratising the plugin-development process. Furthermore, bundling is performed using *Webpack*, as it is considered to be the industry-standard *JavaScript* module bundler, producing significantly minified executables that are optimised for the browser. Finally, the injectable artefact is generated using *Module Federation*, since it synergises as a powerful *Webpack* feature that allows separate *JavaScript* applications to share code and dependencies at runtime, rather than build time. Therefore, it is ideal for the *PlatformOcean* plugin paradigm in creating scalable, independent MFEs.

All elements of the SDK described in Section 5.5 are implemented purely in *JavaScript* using *Node.js*. Functionally, the bundling tool must support four distinct mechanisms. Scaffolding requires writing boilerplate files to a directory, such that plugins are compatible with *PlatformOcean*. This can be achieved using the native node file libraries. Editing requires that any updates to the source code of a prospective plugin are reflected dynamically to the locally-hosted frontend. *Node.js* provides unbridled access to *JavaScript* libraries, enabling the SDK to support *Watchman* with *Hot Module Replacement* (HMR) – two libraries designed specifically for (micro) frontend development. Bundling requires minification similarly to the platform’s client app. Again, implementing the SDK in *Node.js* enables the use of *Webpack*. Finally, publishing is implemented using native *JavaScript* fetch operations.

5.7 Comparison to Existing Extensible Web Architectures

PlatformOcean can be characterised as a web-based plugin architecture that, first and foremost, prioritises flexibility. It achieves this by implementing an intentionally minimal platform kernel that provides lifecycle management, message brokerage, and canonical persistence. This further enables

the platform API to remain deliberately simple while delegating application-level functionality to plugins through contract-level interactions based on arbitrarily structured datagram payloads. This section discusses three other extensible web architectures – *Nextcloud*, *Drupal*, and *Wordpress* – to compare and contrast the various architectural decisions characterising the different pieces of software.

5.7.1 Nextcloud

Nextcloud is an open-source, self-hosted collaboration platform that provides services such as file synchronisation, communication, and productivity applications for building a personal, private cloud. *Nextcloud* therefore effectively serves as an alternative to commercial cloud-storage software such as *Amazon Web Services*, *Google Drive* or *iCloud*.

Unlike traditional content management systems, *Nextcloud* is designed as a modular application platform, where additional functionality can be introduced through independently developed applications (*Nextcloud apps*). These applications extend the core platform, allowing developers to add new functionality while maintaining compatibility with the underlying system. Architecturally, they interact with the core platform through a collection of exposed APIs and service interfaces, which provide access to common functionality such as authentication, storage management, routing, and user preferences. Consequently, applications are able to introduce substantial new capabilities without modifying the underlying *Nextcloud* codebase.

Similarly to *Nextcloud*, *PlatformOcean* supports runtime extension through dynamically installed components obtained from a plugin repository. However, while *Nextcloud* applications become integrated components of an existing application runtime, *PlatformOcean* plugins interact through a minimal coordination substrate, using a runtime composition model to inject independently-deployed frontends that interoperate at the contract level.

Furthermore, *PlatformOcean* offers a philosophical distinction to what a platform actually ‘is’. Whereas *Nextcloud* introduces the design philosophy that a platform can be *extended*, *PlatformOcean* reinterprets the kernel not as the first-class object, but as the minimal set of functionality from which a platform can be *constructed*. In this sense, *Nextcloud* represents an application-centric model of extensibility: a foundational platform is provided initially, and additional applications augment the capabilities *already defined* by the host system. Conversely, *PlatformOcean* adopts a substrate-centric model, where the kernel only provides the plugin-coordination mechanisms necessary for platform construction. *PlatformOcean* is designed with no *a priori* system in mind.

This distinction is particularly important for stakeholder-driven platformisation. Existing extensible platforms primarily empower stakeholders to modify or configure an *existing* digital environment. *PlatformOcean* instead explores whether stakeholders can participate in the ground-up construction of a platform from first principles. It is precisely this user-centric process of selecting, combining, and governing the platform functionality that requires *PlatformOcean* to minimise the assumptions about its kernel. This ensures that domain-specific functionality is an emergent property that manifests through the platform’s plugins, rather than being predetermined.

5

5.7.2 Drupal

Drupal is a free, open-source *content management system* (CMS) and framework written in PHP. As a CMS, *Drupal*’s software enables users to manage the creation and modification of digital content, providing mechanisms for secure access to content, revision control, and reporting [161]. Moreover, as a framework, *Drupal* enables developers to build customised, dynamic web applications. This core system can also be augmented through plugins, enabling a particular module or subsystem to provide functionality in an extensible, object-oriented way [162].

The *Drupal* plugin framework API comprises three core elements. Firstly, *plugin types* define the central controlling class that provides the metadata for how plugins of a given type will be discovered and instantiated. Secondly, *plugin discovery* is the process of finding plugins within the available code base that synergise with a given use case. Lastly, the *plugin factory* is responsible for instantiating the specific plugin(s) chosen for a given use case. Programmatically, using the aforementioned object-oriented paradigm, some *Drupal* module provides an interface, defining the necessary functionality to be performed. This interface is then fulfilled by a given plugin, which implements the prerequisite behaviours in a bespoke way. The controlling module instantiates existing plugins as needed, and calls methods to invoke their functionality [162].

In contrast to *PlatformOcean*, then, *Drupal* implements a fundamentally object-oriented plugin architecture. Extensibility arises through PHP interfaces and abstract classes, meaning that the contract between the platform and plugin occurs at *compile time* – a plugin is recognisable by its method signature, and therefore the interfaces it can fulfil. Consequently, *Drupal* plugins operate within the same runtime environment as the framework and are instantiated and invoked directly by the host application. Conversely, *PlatformOcean* abstracts plugin injection and interaction to the interaction contract layer, allowing plugins to communicate through structured datagrams without requiring knowledge of their internal implementation. A natural symmetry therefore

emerges: in *Drupal*'s object-oriented paradigm, plugin suitability is codified through the methods it can perform, whereas in *PlatformOcean*, plugin suitability is codified through the contract types it can accept. This paradigm shift also repositions the plugin's trust boundary from Section 3.6.3: *Drupal* plugins are reliant on implementation-level execution, whereas *PlatformOcean* plugins are reliant on contract-level execution.

5.7.3 WordPress

Similarly to *Drupal*, *WordPress* operates as a free, open-source CMS that is motivated by democratising website publishing and e-commerce. Accordingly, *WordPress* enables users to build, manage, and publish websites without writing code. This core functionality can then be extended through plugins, which manifest as a package of code that can be installed to provide new features.

WordPress plugins are composed of multiple bundled files, and become compliant system-wide by including a specific front-facing PHP main file with a specifically formatted comment block known as a *DocBlock*. This enables the plugin to become identifiable in the *WordPress* plugin directory.

Architecturally, *WordPress* is similarly motivated to *PlatformOcean*: independently developed artefacts manifesting as plugins are discovered through a well-defined entry point and subsequently injected into the host environment. However, whereas *WordPress* discovers locally installed PHP packages through a plugin *DocBlock*, *PlatformOcean* dynamically resolves remotely-hosted frontend modules via repository metadata and lazy-loading under *Module Federation* [156]. Moreover, *WordPress* adopts the same philosophical approach to extensibility as *Nextcloud*: plugins augment a platform that already defines the fundamental ontology. *PlatformOcean* plugins, conversely, fully compose and define the domain semantics of the platform, since plugins themselves store and define their own state and semantics, without any necessary mutual understanding from the kernel.

5.8 Summary

In summary, this chapter has provided *PlatformOcean*, the *original* implementation of a platformiser for social coordination. This chapter has decomposed the *PlatformOcean* platformiser tool into its three constituent components. The *meta-platform* serves as a plugin repository, responsible for hosting and injecting plugins into platforms, as well as storing preconfigured platform

instances for other communities to use and develop upon. The *platform* serves as the substrate and central node for community interaction, where users can employ the frontend client tool to discover and interface with an ocean of backend platform servers. Each platform stores a history of message and user data, and can be dynamically customised with plugins. *Plugins* are dynamically injectable pieces of remote code generated as part of the *micro-frontend* paradigm. They act as self-contained pieces of functionality that extend the core platform runtime with bespoke functionality. For each of these three components, the relevant algorithms and data structures are specified to demonstrate the full-stack interactions between them.

5

The central design philosophy of *PlatformOcean* has been to view software platforms through the lens of facilitating the self-organisation of social coordination. This perspective has motivated a deliberately conservative kernel design, whereby the expressiveness of a platform arises not from the platform itself, but from the ecosystem of plugin functionalities assembled upon it. Consequently, the kernel is reduced to the minimal coordination substrate required for plugin discovery, lifecycle management, communication, and persistence, while domain-specific behaviour is delegated entirely to its plugins. In doing so, *PlatformOcean* intentionally inverts the conventional core-periphery relationship: rather than treating plugins as second-class accessories to a first-class platform, the platform exists primarily to coordinate the plugins from which it is composed. This design philosophy gives rise to a simple platform mantra: *the platform should prescribe as little as possible, so that communities can prescribe as much as possible*.

Ultimately, existing extensible systems remain platforms: they present a predefined digital environment that can subsequently be customised. *PlatformOcean* instead occupies a different architectural role. It is not itself intended to be the end platform, but rather the mechanism by which platforms may be assembled. This is precisely why we characterise *PlatformOcean* as a *platformiser*: software whose primary purpose and principal motivation is to facilitate the stakeholder-driven process of both platform formation and reformation, and conduct the process of *ethical platformisation* within the necessary guardrails explored in Chapter 2.

Beyond the components, this chapter has also specified the *PlatformOcean* API that is codified in a wrapper component and attached to the dynamically-injected plugins to achieve the distributed message-passing capabilities. The API consists of a set of methods that plugin developers call to enable arbitrary datagrams to be sent from any given plugin, across a platform instance, to a matching plugin on a separate machine. The API also provides two datagram dissemination protocols: an *intra*- and *inter*-plugin protocol for sending messages across plugins of the same type, and extending plugins of different types respectively.

Finally, we have detailed the *OceanBundle* SDK, a CLI tool for third parties to develop plugins which are compatible with any *PlatformOcean* platform. This SDK supplies four methods – scaffolding, editing, bundling, and publishing – that guide programmers through the four stages of the plugin development lifecycle.

A critical facet of the *PlatformOcean* architecture is that plugins are *platform agnostic* – the same plugin can be reproduced across any platform instance in the ecosystem. The ‘meaning’ of the plugin is derived purely at runtime through its *instanceID* and list of *dependencies*. Cross-plugin communication (see Section 5.4) is declared at a contract level, meaning that the dependencies can be inferred at runtime from the IDs of the other plugins on the platform.

Having demonstrated through *PlatformOcean* that a *platformiser* is *implementable*, the next step to demonstrating viability is stress-testing the tool across multiple testcases. The following chapter demonstrates, through six testcases, that *PlatformOcean* is *applicable*.

6

Platformiser Applications

6

Contents

6.1 Introduction	113
6.2 Case Study 1a: Twitter-like	115
6.2.1 Functional Requirements	116
6.2.2 Implementation	116
6.3 Case Study 1b: Pong-like	118
6.3.1 Functional Requirements	118
6.3.2 Implementation	119
6.4 Case Study 2: WhatsApp-like	121
6.4.1 Functional Requirements	122
6.4.2 Implementation	122
6.5 Case Study 3: Sporting Association Coordinator	122
6.5.1 Functional Requirements	123
6.5.2 Implementation	123
6.6 Case Study 4a: Mind my Mind	126
6.6.1 Functional Requirements	126
6.6.2 Implementation	127
6.7 Case Study 4b: Climate-Smart School Meals Advisor	129
6.7.1 Functional Requirements	129
6.7.2 Implementation	130
6.8 Quantitative Profiling	133
6.9 Summary	136

6.1 Introduction

Having specified an abstract platform representation through a set of high-level components, and subsequently formalised them into a concrete platform implementation, this section charts the evaluation of the platform through a series of prospective case studies.

As a distributed client-server model, this platform has the capacity to scale in two dimensions: both in *platform cardinality*, as the number of platform instances used in the application increases, and in *plugin cardinality*, where the number of (different) plugins used for each platform instance increases. Figure 6.1 specifies six prospective applications across these two dimensions.

		Platform Cardinality	
		1	... → ... N
Plugin Cardinality	1	Case Study 1a – <i>Twitter-like</i>	Case Study 2 – <i>WhatsApp-like</i>
	...	Case Study 1b – <i>Pong-like</i>	
	↓		
	M	Case Study 3 – <i>SAC</i>	Case Study 4a – <i>Mind My Mind</i> Case Study 4b – <i>CSSMA</i>

Table 6.1: *PlatformOcean* implementation examples – the viability of the platform architecture is further evaluated through six examples ranging from single-platform, single-plugin, to multi-platform, multi-plugin

- 1a. *Twitter-like* – A simple single-platform, single-plugin system where users can subscribe to a single ‘blog’ and send image- and text-based datagrams;
- 1b. *Pong-like* – A more advanced single-platform, single-plugin system requiring the dissemination of multiple, high-frequency state-representing datagrams to play *Pong*. This tests both the message-frequency tolerance of the system, and ability to abstractly represent plugin states;
2. *WhatsApp-like* – A multi-platform, single-plugin system analogous to *WhatsApp* group chats: users can send and receive image- and text-based datagrams across different platform instances;
3. *Sporting-Association Coordinator* – A single-platform, multi-plugin system used for the coordination of a (legitimate) local sports team: users organise match fixtures, carpooling, laundry rota, and message exchange;
- 4a. *Mind My Mind* – A multi-platform, multi-plugin system used to coordinate (legitimate)

students in a university. Plugins are designed based on the *Bartle taxonomy of player types*, allowing student groups to tailor their platform to fit their characteristics; and

- 4b. *Climate-Smart School Meals Advisor* – A more advanced multi-platform, multi-plugin system allowing for schools to coordinate the sourcing of local ingredients, and planning of sustainable school meals. Plugins allow for food-source database management, menu curation, and (AI-driven) menu recommendation. Platforms can be instantiated both in individual schools for meal planning, and multiple countries/constituencies for database management/availability.

This thesis has demonstrated that large-scale, ground-up self-organised collective action, which properly empowers stakeholder communities, requires a new type of software tool. We have called this tool a *platformiser*. In order to facilitate such self-organised collective action, this platformiser ought to be able to replicate stereotypes of social media interaction, such as a *Twitter-like* and *WhatsApp-like* (cf. [69]).

The purpose of the *PlatformOcean* platformiser is more than just to reimplement existing social media applications as a stakeholder-owned distributed system. A principle consideration of the tool is to demonstrate *reconfigurability*, and to compose meaning from a set of plugins into a single, cohesive platform. Therefore, given that the platformiser can scale in two orthogonal dimensions – plugin and platform cardinality – we also evidence platform of increasing complexity through the *SAC*, *MMM*, and *CSSMA* testcases.

While these testcases showcase that the platformiser is sufficiently expressive to implement multi-plugin, multi-platform systems, they must also investigate the technical constraints of the system. The tolerance to high-frequency message passing is therefore evidenced in the *Pong-like* testcase.

In summary, these testcases are curated to examine the platformiser *functionally*, through its ability to reproduce stereotypes of social media interaction; *normatively*, through its ability to express high-complexity systems; and *performatively*, through its tolerance to high-frequency message dissemination.

6.2 Case Study 1a: Twitter-like

The simplest manifestation of a *PlatformOcean* platform is as a single instance, with a single-plugin. One such use-case is a *Twitter-like* (currently *X-like*) platform, where multiple users subscribe to

a single instance (akin to a centralised *Twitter* webpage), and interact through a single plugin to mimic a series of ‘blog posts’. The analogy, then, is that each member of the platform instance represents a *Twitter* user, and each (many-to-many) datagram sent over the platform represents a *Tweet*. This analogy can be developed further to consider multiple *Twitter-like* platforms as distinct *Twitter* sub-spheres/micro-environments, each fostering their own bespoke communities. In this vein, the idea of a *Twitter-like* platform can even be repurposed as a *Reddit-like* platform, where each decentralised instance represents a *subreddit*. We formalise this operation through the following functional requirements:

6.2.1 Functional Requirements

By abstracting the core functionality of *Twitter*, users of a *Twitter-like* platform must be able to...

- *Broadcast*: ...send an arbitrary datagram (i.e., a multimedia message) across a platform instance; and
- *Update*: ...edit/delete an arbitrary datagram, given that they have access privileges (i.e., sharing the relevant provenance).

6.2.2 Implementation

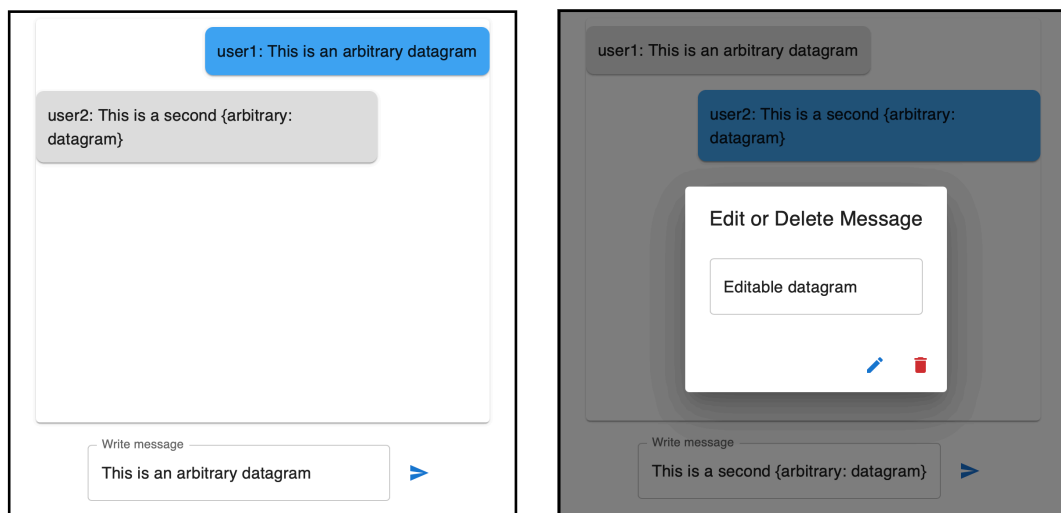
The minimal functionality of a *Twitter-like* platform enables the functionality to be codified in a single plugin, shown in Figure 6.1. Operationally, users must have a mechanism for identifying message ownership (through an author ID), transmitting arbitrary content through that message, and performing different protocols. An appropriate datagram is shown below:

```
{  
  "nickname": "user1",  
  "authorID": "UUID",  
  "content": {  
    "messageString": "This is an arbitrary datagram",  
  }  
  "protocol": "CREATE"  
}
```

However, given that *PlatformOcean* leverages the (arbitrarily) *serialisable* datatype for datagram transmission, additional state-modifying data can be codified, as below:

```
{
  "authorID": "SENDER_ID",
  "nickname": "USER",
  "payload": {
    "topic": "TWEET_TOPIC",
    "content": "TWEET_CONTENT",
    "metadata": {
      "maxLength": 280,
      "timeToLive": 3600
    }
  }
}
```

if, for example, the plugin designer desired that messages have a maximum length of 280 characters, or an ephemeral duration of one hour.



(a) *Blogpost* plugin window (1) – messages are conditionally rendered, both in colour and selectability, depending on the user. This window shows the ‘user1’ perspective

(b) *Blogpost* plugin window (2) – authors of a message can select it, prompting editing or deletion. This window shows the ‘user2’ perspective

Figure 6.1: Windows of plugin used to address the *broadcast* and *update* functional requirements for the *Twitter-like* plugin

Furthermore, this structuration permits a further ‘conversion’ of the basic, broadcasted blog into a stratified *Tweet*-oriented platform. For example, messages can be routed/multiplexed by

topic to create a (recursive) ‘sub-*Twitter*’. In this formulation, the functionality displayed in Figure 6.1 can be repurposed, such that the *Twitter-like* platform consists of multiple *sub-Twitter-likes* designed for a specific blog topic. Accordingly, the single *Twitter-like* platform becomes analogous to a *Reddit-like* platform.

6.3 Case Study 1b: Pong-like

Pong is one of the earliest arcade video games, simulating table tennis. Players control paddles to hit a ball back and forth, trying to score points when the opponent misses. Similarly to the *Twitter-like* platform, the *Pong-like* platform therefore manifests as a single-platform, single-plugin – two users of a single platform instance interact through the *Pong* plugin, each controlling one paddle.

Where these two platform delineate, however, is through the frequency of message distribution. Whereas, the *Twitter-like* platform functions as a low-frequency message relay, distributing messages only at the rate at which users compose them¹, the *Pong-like* platform must constantly relay datagrams across the network to maintain a consistent state representation across all connected clients. This state-dissemination rate can therefore be (reasonably) unbounded: 30 updates a second, say, for a 30FPS game. Accordingly, state consistency is guaranteed through the *PlatformOcean* message queues, allowing plugin developers to constantly track the size of queue and use the *pollMessageQueue* method from the API.

6.3.1 Functional Requirements

In order to play a *PlatformOcean*-ised version of *Pong* (that tried to replicate the original *Atari* version as closely as possible), users must be able to...

- *Registration*: ...enter into a ‘waiting room’, pairing them up with another player before the game can start; and
- *Gameplay*: ...move a single paddle, either on the left or right of the screen, up and down to hit a virtual ball.

¹this assumes a conservative pool of users for the proof-of-concept version of *PlatformOcean*

6.3.2 Implementation

This platform testcase serves to replicate the ‘classic’ *Atari* game system of the 1970s. Accordingly, the platform instance becomes analogous to the *Atari* console, and the plugin implements the game. Therefore, the single-platform, single-plugin model is most appropriate for this implementation, as with the *Twitter-like*.

In the *Twitter-like* platform, each user authors independent messages to contribute to the blog. The global, synchronised state emerges through aggregation, wherein multiple messages can coexist without conflict because they do not compete to define a single shared truth.

Programming a game through distributed datagrams requires a different paradigm. Unlike with the *Twitter-like* platform, state synchronisation is effectively ‘privileged’. Whereas with the *Twitter-like* platform states could converge without conflict, there is only one ‘true’ gamestate in *Pong*. Any conflicting gamestates present on the different clients *cannot* converge. The position of the *Pong* ball, for example, cannot simultaneously differ across clients without breaking the game’s logic. Messages authored by user *a* on the *Twitter-like* platform are ‘owned’ by user *a*, with similar logic applied to messages authored by user *b*. The *Pong* ball, however, is effectively ‘owned’ by all users.

Consequently, synchronising the gamestate in a *Pong-like* platform requires role delegation: some user must determine and enforce the single ‘true’ state. To achieve this, we delegate state synchronisation to the *host*, shown in Figure 6.2b. Datagrams distributed by the host then codify the state as follows:

```
{
  "type": "gameStateUpdate",
  "gameState": {
    "newX": 300,
    "newY": 100,
    "newSpeedX": 1,
    "newSpeedY": -1,
    "scoreLeft": 4,
    "scoreRight": 6
  }
}
```

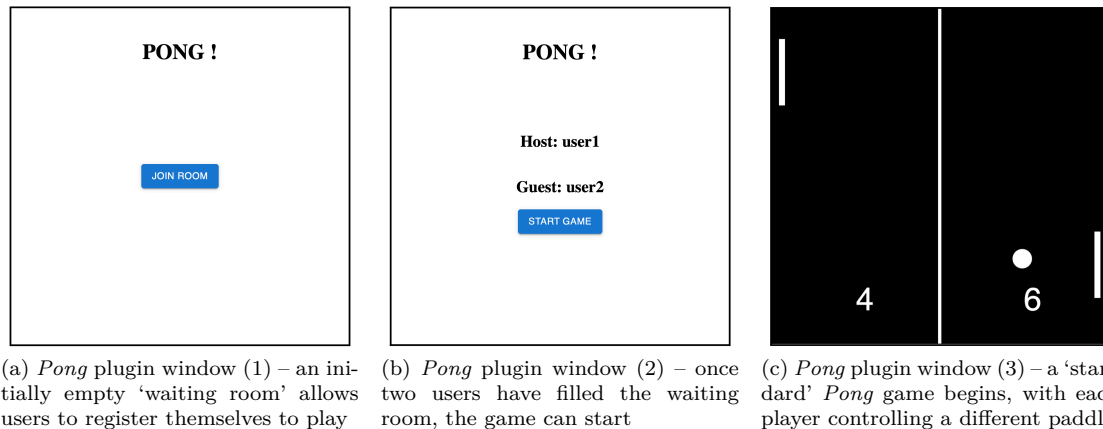


Figure 6.2: Windows of plugin used to address the *registration* and *gameplay* functional requirements for the *Pong-like* plugin

6

with appropriate conditional logic codified into the plugin, such that datagrams are only distributed if the client has host privileges. The position of the non-host’s paddle, however, is ‘owned’ by the non-host, allowing a separate communication pathway to sit on top of the gamestate synchronisation datagram:

```
{
  "type": "paddleRightUpdate",
  "paddleRight": 100
}
```

alongside an identical implementation for the host’s paddle. The specification of a *contractType* parameter to the datagram allows for demultiplexing (and conditional handling) during plugin development (see Listing 5.1). The three screens used to implement this plugin are shown in Figure 6.2, demonstrating the waiting room, ready screen, and gameplay screen.

Conventional authoritative-server systems accept ‘requests’ from the players and compute and simulate the resulting gamestate from those requests. For example, players one and two may submit a *movePaddle* request, and the server will be responsible for calculating the velocity and position of the ball, redistributing this authoritative information, and the connected clients will simply render what the server informs.

To comply with the *PlatformOcean* paradigm, however, the *Pong-like* platform’s state representation and management are implemented at the contract level. This means that the platform leverages an authoritative *host* to enforce semantics and ownership over the gamestate, combined with an authoritative *server* to enforce delivery and replay. This reconfiguration arises because

the server can *route* the contracts, but cannot verify the gamestate since it cannot *understand* the contract semantics. There is therefore a tension between intentionally decoupling the platform and its plugins, and positioning the platform kernel as the authoritative state machine. Making the platform host responsible for the state dissemination introduces opportunities for cheating, say, as the contract could simply propose to increment player one's score whenever they feel like it.

Although delegating gamestate dissemination to the authoritative host introduces opportunities for malicious behaviour, the platform mitigates this through deterministic contract validation and replayable event history. Specifically, the server maintains an authoritative ordered log of the previous (N) state transitions and disseminated messages, enabling connected clients to independently reconstruct and validate the contract state machine. In this way, the paradigm moves away from the host merely being a trusted entity, and instead towards an *optimistic* entity that can be audited *post hoc* using the authoritative server logs.

Each contract in this platform can then be considered as a *Markov* process: each contract encodes a state-updating message which can be applied to the current state using the plugin's internal state machine to produce the next state:

$$State_{n+1} = f(State_n, Contract_n)$$

where, crucially, this state machine is identically implemented across all clients (since all clients instantiate the exact same plugin). Semantic correctness therefore comes from the ability of participants to deterministically replay and verify the transition history against the contract's validation rules, thanks to the correctness guarantees from the platform kernel.

Ultimately, this approach leverages both sides of the parallel trust streams described in Section 3.6.3. Technologically, clients can independently verify the state transitions provided by the host and authoritative platform kernel log to detect cheating. At the same time, socio-politically, clients should also exercise their right to legitimate consent for disassociating from a platform: *you never have to play a gentleman's game with a cheater*.

6.4 Case Study 2: WhatsApp-like

A *WhatsApp-like* platform is the natural extension of a *Twitter-like* platform. Rather than multiple users interacting over a single platform instance, having multiple users interacting over *multiple*

platform instances effectively recreates the *WhatsApp* group-chat system. Rather than being an admin of a *WhatsApp* group chat, users can instantiate (and serve as the admin of) a *PlatformOcean* *WhatsApp-like* platform. Moreover, rather than joining a group chat, users can register themselves to a *PlatformOcean* platform instance. Figure 3.3 demonstrates this contrast.

6.4.1 Functional Requirements

Similarly to the *Twitter-like* platform, users of a *WhatsApp-like* platform network must therefore be able to...

- *Broadcast*: ...send an arbitrary datagram (i.e., a multimedia message) across a platform instance;
- *Update*: ...edit/delete an arbitrary datagram, given that they have access privileges (i.e., sharing the relevant provenance); and
- *Voluntarily Associate*: ...give consent to register to, and revoke consent to deregister from, a given platform instance.

6.4.2 Implementation

Given that this platform testcase repurposes the plugin defined in Figure 6.1, we deliberately avoid rewriting the implementation from Section 6.2.2.

6.5 Case Study 3: Sporting Association Coordinator

Hittenhope is a (genuine) football club that arranges an informal 5-a-side football match once a week, at a regular time-slot at a local Soccer Centre (a private operator of dedicated football pitches for hire). Teams are selected according to whoever is available that week; but a judicious selection is required to ensure a ‘fair’ and ‘balanced’ game that is close and enjoyable for everyone. Beyond notification of availability and team selection, there are various other activities associated with a game, such as travel and kit maintenance (i.e., laundry).

6.5.1 Functional Requirements.

An informal ‘player-centred’ participatory design process identified the following functional requirements. Players should be able to...

- *Squad management*: ...add and remove people from the squad (i.e., the pool of possible players);
- *Availability*: ...notify that they are available for selection in a particular week;
- *Team recommendation*: ...ask for two ‘fair’ teams to be automatically recommended, selected from those who have indicated their availability;
- *Metriation*: ...rate each other privately, and rate specific games according to their personal perception of ‘fairness’, providing data for automated team recommendation;
- *Communication*: ...engage in multi-logue ‘chats’ as per any other social media instant messaging system;
- *Participation*: ...visualise fairness in ‘support’ activities, such as kit maintenance (e.g., laundry); and
- *Coordination*: ...minimise expense, emissions, and parking pressure, by forming car pools for available players.

Taking the meme “there is an app for that” seriously for a moment, each one these separate functional requirements could be satisfies by a single app. However, a far better approach would be to incorporate each disparate app as a *micro-app* that can be plugged in and integrated within an over-arching platform. *PlatformOcean*’s plugin architecture specifically targeted at supporting such an approach: the design principles outlined in Section 5.4 enable each required plugin to be dynamically added, allowing the platform development to be validated against the functional requirements.

6.5.2 Implementation

Given that the *squad management*, *availability*, *team recommendation*, and *metrics* requirements are inextricably linked, we implement all four within the single, *tabbed* plugin shown in Figure 6.3.

The *genericity* of the plugin architecture and development toolchain are both leveraged to produce this plugin. Most interface elements use the *React* (MUI) framework, combined with a bespoke grammar for state management and data representation. This is achieved using the external library support from the *Webpack bundler*, and arbitrary *JsonNode* representation supported by the platform, respectively.

A plugin designer must consider the plugin grammar both from the perspective of sending and receiving datagrams. This grammar must be defined such that each component of this plugin is uniquely characterised, and the state is synchronised across all platform instances.

The *squad management* component of the plugin has two functions - adding/removing players from the squad, and submitting a rating of these players. Accordingly, this plugin should produce two messages: firstly, a means of identifying a player. For reusability (to minimise repeated data in the platform), we also attach the player's availability to this datagram, so that it is useable in the *teach recommendation* component:

```
{
  "type": "playerState",
  "content": {
    "name": "Player1",
    "state": "PLAYER_STATE.Unavailable"
  }
}
```

This plugin component then dynamically re-renders in response to the message. This particular implementation defines a custom *React hook* to extract the relevant messages from the message history, and maps them from a list of datagrams to a list element.

Secondly, this component must produce messages that represent the rating of a squad. Again, we leverage the arbitrary *JSON* format to uniquely characterise the message type, and use a list of objects to codify the ranking:

```
{
  "type": "playerRanking",
  "ranking": [
    {"name": "Player1", "score": 3},
    {"name": "Player2", "score": 1},
  ]
}
```

```
    {"name": "Player3", "score": 5},  
    ...  
  ]  
}
```

which is also compatible with the custom *React hook* described above, allowing the *team recommendation* plugin to prefetch the relevant messages, preprocess them for input to the matchup generation algorithm, and render the output.

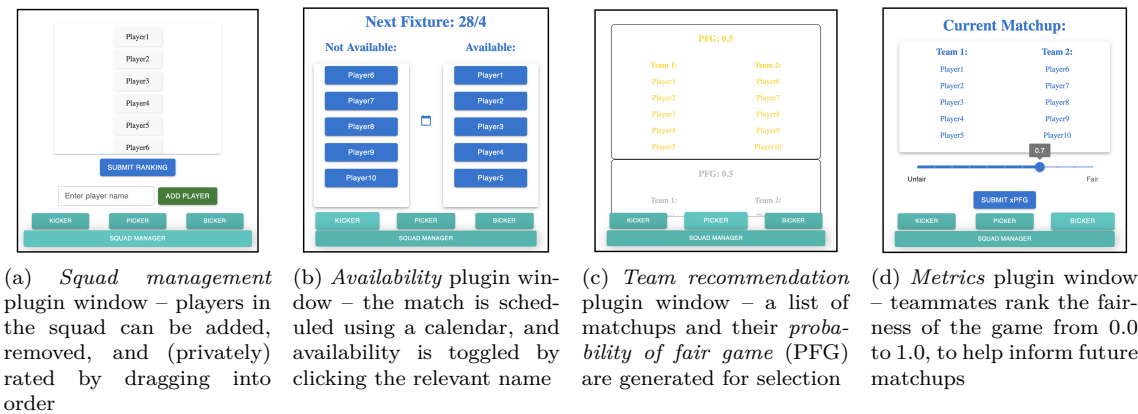


Figure 6.3: Windows of plugin used to address the *squad management*, *team picker*, and *management rater* functional requirements

Similarly, we provide the implementation of the *chat app* in Figure 6.4a. This plugin leverages the functionality of the wrapper component [109] to dynamically inject the name of the registered user into the sent datagram. This history of datagrams is then conditionally rendered to the UI, where a grey, left-aligned styling is applied to messages that originate from a different user instance (identifiable by the *sender* parameter in Table 5.3), and a blue, right-aligned styling otherwise.

Furthermore, we address the ‘fair’ *participation* functional requirement with the custom plugin shown in Figure 6.4b. To codify this requirement, we envision a list of datagrams comprising a mapping of player name to laundry count. Here, the players’ names can be entered and removed to display a square grid, with each name superimposed on a button with colour (‘heat’) proportional to the frequency of laundering.

Finally, to address the *coordination* functional requirement, we provide a custom car-pooling plugin in Figure 6.4c. This plugin allows players to enter their name and address to produce a waypoint. This list of waypoints is then rendered to the UI as a *draggable*, such that users can identify the start and end locations of the route. Pressing *generate route* submits a request to the *Open Route Service API*, to produce a route.

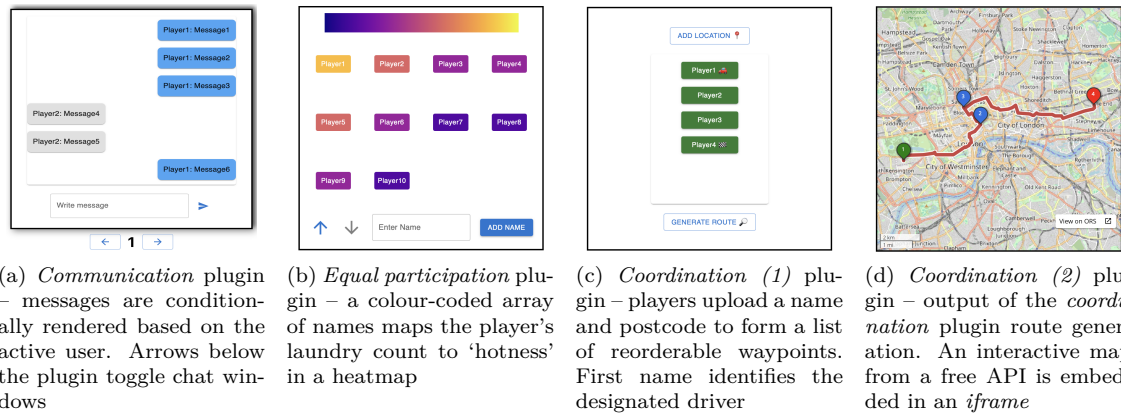


Figure 6.4: Plugins used to address the *chat app*, *laundry rota*, and *carpooler* functional requirements

6

6.6 Case Study 4a: Mind my Mind

In the context of the data economy [31], the idea of privacy as the right not to be observed has led to the development of PET (Privacy Enhancing Technology). More recently, in the context of the so-called attention economy, the idea of attention as the right not to be interrupted has led to analogous proposals for *Attention Enhancing Technology* [163]. This is especially important for contemporary students, who have to focus on coursework and examinations when an induced dependence on mobile devices and social media, distracting their focus and siphoning their cognitive energy, is a deliberate factor in product and service design. Therefore, we propose to develop a platform for personalised, self-organised, mutually-supportive study that promotes concentration and preserves attention.

6.6.1 Functional Requirements

The primary motivation for the Group Study platform is support for collective study, where small groups of students can coordinate their time according to their preferred study method. An informal 'student-centred' participatory design process identified the following functional requirements. Students should be able to...

- *Selection*: ...adjust a dynamically-adaptive task management systems according to their own diverse preferences and study habits;
- *Feedback*: ...exploit 'internal' mechanisms for incentivising themselves and each other, whereby

study is its own immediate reward, but provides longer-term reward in improved performance and grades; and

- *Reward*: ...enjoy the benefits of an ‘external’ reward system that provides feedback and positive reinforcement.

From the platform developers perspective, techniques to satisfy these demands include the use of the *inner voice*, or *inner speech* (endophasia) [164] for its role in productivity and cognitive functions such as memory, decision-making, and emotional regulation; and a gamified system leveraging Bartle’s Taxonomy of Player Types [12]. This taxonomy classifies gamers into four archetypes: *achievers* (motivated by rewards), *explorers* (motivated by discovery or statistics), *socialisers* (motivated by connection), and *killers* (motivated by competition).

However, this presents a different challenge to the previous case study, as the mapping between functional requirements and plugins is no longer one-one. This demands that the implementation requires multiple plugins for each requirement, in particular to support the full range of gamer types.

6.6.2 Implementation

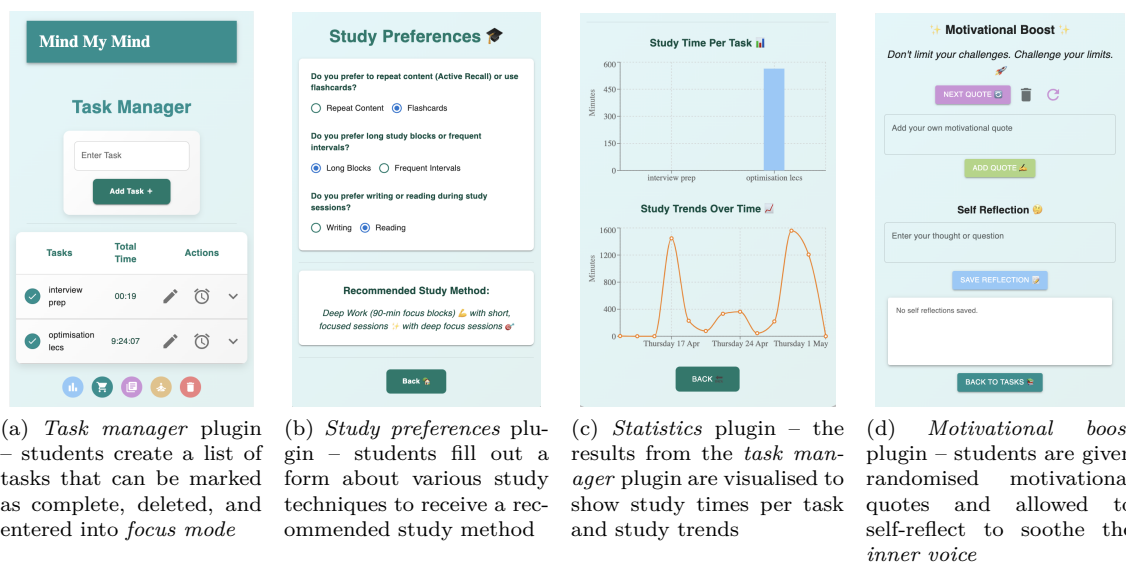


Figure 6.5: *Task manager*, *study preferences*, *statistics*, and *motivational boost* plugins for the *Group Study Coordinator* platform

As a baseline requirement, the *Group Study Supporter* platform acts as a task manager for coordinating the students’ time. Figure 6.5a shows the UI of the *task manager* plugin. Here,

students enter the name of the task into the form, whereby it is added to a list of outstanding tasks. This list provides each task with its personal timer, showing how long the task has been underway, and a set of further actions, for controlling the state of a task – i.e., editing the name of, or deleting the task. Critically, each task comes with a *focus mode* (represented in Figure 6.5a by the clock icon), wherein the student is taken to a minimalistic, timer-based UI to record how long they have engaged in uninterrupted study. This promotes focus by removing external distractions.

In conjunction with the *task manager*, the *Group Study Coordinator* provides a *study preferences* plugin to help customise the interface of the *task manager*, based on a set of preferences and study habits. Figure 6.5b illustrates the form used to record user feedback in tailoring the *task manager* interface. This satisfies the *selection* functional requirement.

6

Furthermore, a mechanism for recording a student's progress is provided by the *statistics* plugin, shown in Figure 6.5c. Here, a histogram records the study time per task, and a line graph records the average time spend studying each day to identify trends in students' studying habits. This mechanism addresses the *feedback* functional requirement, and particularly supports the *explorer* archetype, who are motivated by the observation and discovery of trends and statistics.

Finally, to address the *reward* functional requirement, wherein students are provided with positive reinforcement for their studying, Figure 6.5d shows the *motivational boost* plugin. Figure 6.5d. The role of this plugin is twofold: firstly, inspirational quotes are randomly generated (from a database) to boost the students' morale. Secondly, the *motivational boost* plugin calms the inner voice by acting as a journal, allowing students to record thoughts for self-reflection.

Beyond the 'core' components shown in Figure 6.6, we also provide a repository of further plugins to help customise a bespoke study platform. In particular, there are plugins for quizzes to get feedback to appeal to *explorer* types. There is also a chat-app plugin and kudos system to appeal to *socialiser* types. Here we can re-use the chat-app from Section 6.5, demonstrating how different platforms can exploit the same plugins albeit for different purposes. To appeal to the *achiever* type, a plugin can be added to let students earn 'coins' (a conceptual, gamified currency) to spend in a 'shop', giving options for customisation, say. Finally, to appeal to the *killers*, we provide a leaderboard plugin which records the data from the *statistics* plugin and compares it against other students' performances, to encourage competition.

A platform can be defined as a set of components. This set comprises low-variety, core components and high-variety peripheral components. The low-variety core provides the base functionality and is relatively stable, while the high-variety peripherals offer customisation and variation [2]. In

this paradigm, the *task manager* and *motivational boost* plugins are low-variety, core components, which are instantiated on the ‘vanilla’ *Group Study Supporter* platform. The *statistics*, *chat-app*, *quizzes*, and *leaderboards* plugins, then, serve as the high-variety *complementary* components, allowing for bespoke customisation of the pre-instantiated platform.

6.7 Case Study 4b: Climate-Smart School Meals Advisor

An estimated 418 million children benefit from school meals worldwide in at least 176 countries from all income levels, contributing to an annual global investment of US\$48 billion [165]. School meal programmes (SMPs) promote children’s health, nutrition and wellbeing outcomes, and improve educational attendance and attainment [166]. SMPs have also been recognised as a potential pathway towards environmental sustainability by driving demand for nutrient-rich, locally produced, diverse foods that can deliver on nutrition and health goals for children [167].

School menu planning is one strategy that can strengthen the implementation and sustainability of SMPs. Well-planned menus ensure that school meals meet children’s dietary requirements, reducing under-nutrition and micronutrient deficiencies [168]. They enable the efficient use of limited resources by optimising food choices based on cost, seasonality, and availability—improving financial and logistic sustainability [169].

Despite this, to date, no school menu planning tool offers a tool for sustainable school menu planning that enables local actors to retain agency over data management, statistical analysis and reasoning. Moreover, monolithic platform approaches fail to deliver the decision-making facilities to develop context-specific, adaptive, and climate-smart school menus. Therefore, this platform testcase serves to develop a decentralised school menu planning tool that supports the delivery of nutritious, healthy, and equitable SMPs within regional variation. This tool is called the *climate-smart, school meals advisor* (CSSMA).

6.7.1 Functional Requirements

To achieve these goals, *CSSMA* platform instances *must* enable users to...

1. *Meal Creation*: ...curate a ‘basket’ of ingredients to design a meal (i.e., a set of ingredients for a given recipe);

2. *Nutrition Visualiser*: ...visualise, both for individual ingredients and full meals, the nutritional information (e.g., protein, carbohydrates, and micronutrients) and ecological information (e.g., greenhouse gas emissions and water footprint) of the foods; and
3. *Menu Builder*: ...use the meals specified in the *meal creation* function to curate a five-day menu, representing the meals served in a standard school week;

and *should* allow users, at their behest, to augment the ‘core’ platform to...

4. *Database Management*: ...dependent on access privileges, add, update, and delete ingredients (and their corresponding nutritional information) from a centralised database; and
5. *Meal Recommendation*: ...specify a series of nutritional requirements and list of potential ingredients, and generate a series of meals that fulfil the criteria.

6

6.7.2 Implementation

Two crucial components of *PlatformOcean* are: one, its role of a platformiser in producing an ecosystem of distributed platform instances; and two, the capacity for the platforms produced by the platformiser to be customised, at runtime, using its plugin architecture. This makes *PlatformOcean* the perfect candidate for this project – various school districts can host their own, localised platform instances and interface with the databases for their respective regions. Local authorities can duplicate a school district’s platform instance, add the database management plugin, and accurately compose the available ingredients in the area. Each school then has sovereignty over its own platform instance, which the students (and staff) can use. This project then becomes *scalable*, as *PlatformOcean* promotes generativity through the repurposing of platform instances: other school districts can duplicate an existing platform, curate its set of plugins and database usage, and produce a bespoke SMP tool.

To address the functional requirements for *CSSMA*, we envision a multi-platform, multi-plugin approach. A ‘core’ platform distributable provides only the *menu builder* functionality, however this platform can be augmented depending on the use-case – local governments can duplicate the platform and attach the *database management* tool to curate the available ingredients list, and other school districts may find use for the *meal recommendation* tool, thereby tailoring the platform to their particular needs through customisation via plugins.

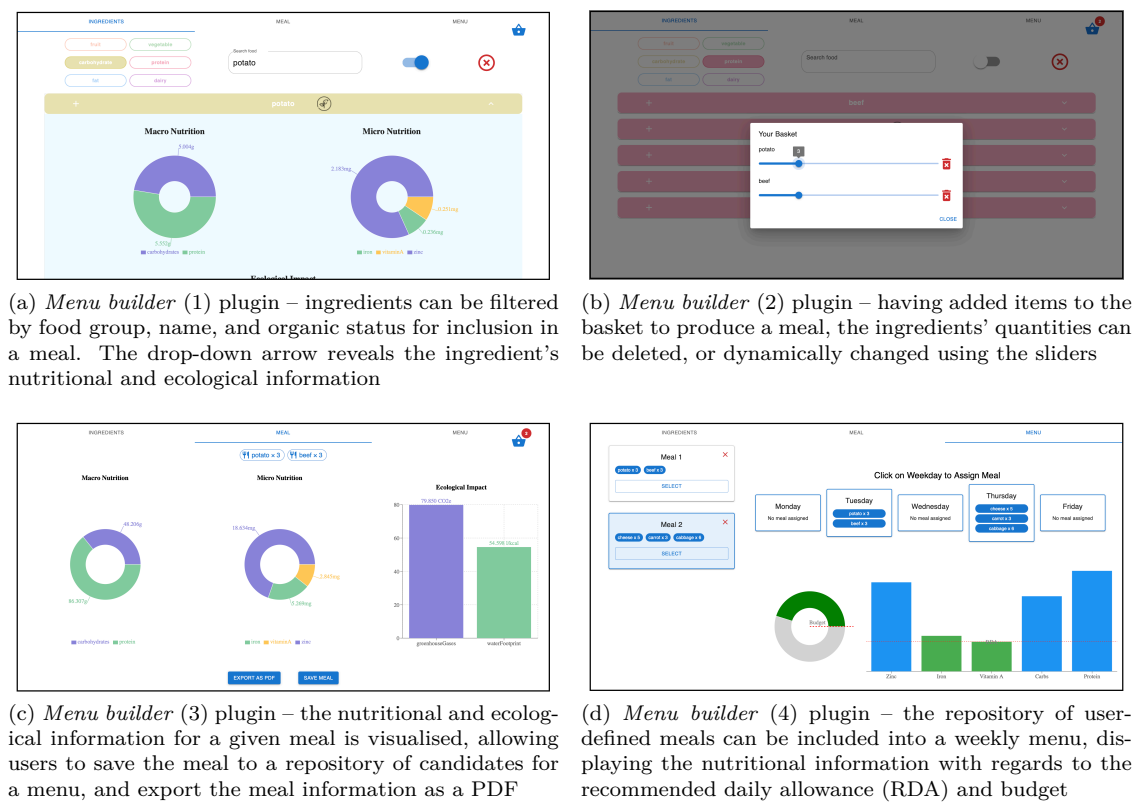
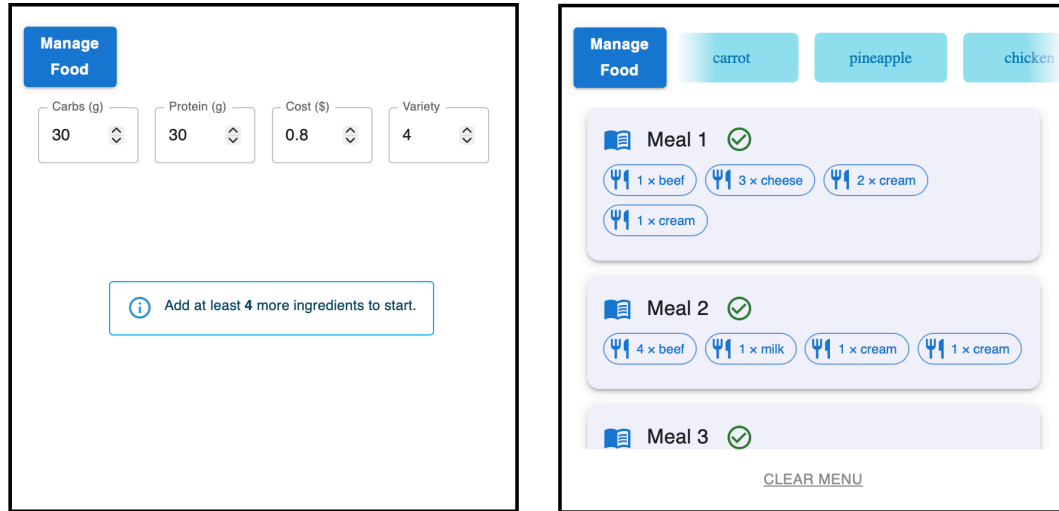


Figure 6.6: Ingredient management, and nutrition visualiser; meal builder, and nutrition visualiser; and menu curator components of *menu builder* plugin, used to address the *meal creation*, *nutrition visualisation*, and *menu building* functional requirements

Figure 6.6 shows the core platform distributable: a full-screen plugin with three pieces of interconnected functionality. Having specified the appropriate database backend (i.e., a region-specific connection defined on platform boot), users gain access to the ingredient list. Each ingredient in the list provides a drop-down region to visualise the micro- and macro-nutrient information, as well as the ecological footprint. Ingredients can subsequently be added to a basket, shown in the top-right of all screens in Figure 6.6. Clicking on the basket then provides a second menu, shown in Figure 6.6b, for dynamically adjusting the quantity of the ingredient. The plugin function in Figure 6.6c then aggregates the nutritional and ecological information of the ingredients in the basket using Algorithm A.1 to produce a full meal that can be saved (for inclusion in a menu) or exported as a PDF. Finally, Figure 6.6d shows the menu creation functionality: meals generated using the previous tool can be allocated to a day of the week, recording the allowed budget, and nutritional targets.

Beyond the core, *menu builder* plugin, the *CSSMA* platform offers a repository of additional plugins to augment the platform. Rather than manually experimenting with ingredient quantities to reach the RDA for a given meal, users can leverage the *meal recommender* tool to perform a

constrained optimisation search over all possible ingredient combinations from a given subset to address the nutrient requirements. Reductively, this search serves as a knapsack algorithm that generates all possible quantities of some ingredient combination that meets the requirement. We formalise this algorithm in Algorithm 4 and visualise the result of a potential search in Figure 6.7.



(a) *Meal recommender* (1) plugin – users are initially required to specify nutritional constraints and add a basket of ingredients for the prospective meal. Given a *variety* constraint of n ingredients, an ‘info’ prompt requires that n ingredients are available

(b) *Meal recommender* (2) plugin – having met the ingredient constraint, a recursive search algorithm is performed to find a list of suitable meals. A carousel of chosen ingredients cycles at the top of the plugin

Figure 6.7: *Meal recommender* plugin to address the *meal recommendation* functional requirement

A *CSSMA* platform instance is not intended as a standalone tool, but rather as a node in a larger ecosystem of sustainable meal planning. Within this ecosystem, we envision the possibility for various different roles, reflecting the different hierarchical levels in school administration. At the school district level, teachers and students can collaborate with chefs using the *menu builder* plugin to suggests ingredients for school meals and menus. Furthermore, if the chefs need computational support to help work out the appropriate ratios of the ingredients to meet a set of nutritional requirements, they can employ the *meal recommender* plugin.

Common to both of these plugins, is a centralised database recording the nutritional and ecological data for the different ingredients. To ensure the validity, accuracy, and consistency of this database, we delegate the maintenance to a central authority – the local council, perhaps – who can use their own platform and the *database management* plugin shown in Figure 6.8. Given the relevant access rights, this tool permits the local authority to specify the address of a *CSSMA* database, and update the repository of ingredients accordingly.

In this way, *CSSMA* operates not only on a school or district level, but has the capacity to

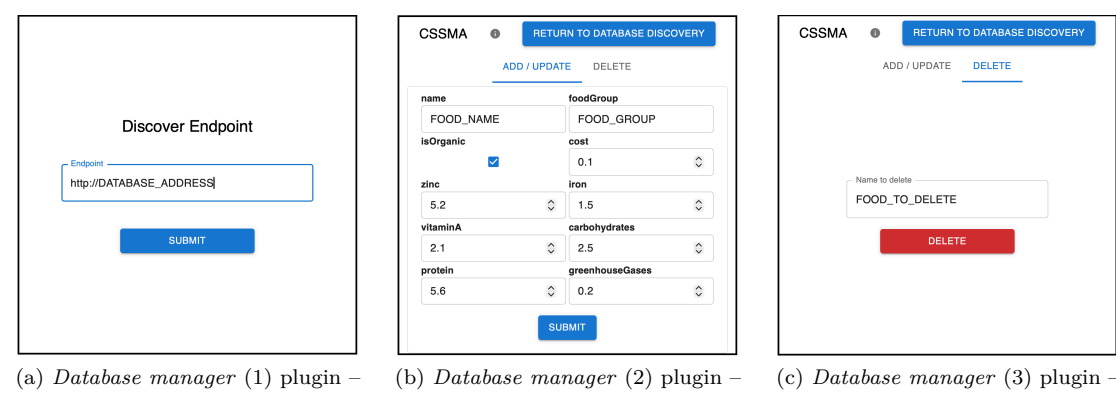


Figure 6.8: Registration, submission, and deletion options for the *database manager* plugin, used to address the *database management* functional requirement

scale far beyond. The modularity and distributed design of *PlatformOcean* means that the backend defined in Figure 6.8 can be deployed to curate ingredient data at varying levels of granularity – whether borough, city-wide, or national. Consequently, the system supports both local autonomy and broader standardisation, depending on institutional requirements.

6.8 Quantitative Profiling

PlatformOcean allows for the arbitrary specification of plugins using language primitives, external libraries, and the wrapper API defined in Table 5.4. Plugins then specify arbitrary interaction contracts of the type defined in Section 4.3.3 to define custom grammars and datagram schemas for distributed messaging.

In this way, *PlatformOcean* can perform its own quantitative profiling by specifying the appropriate plugin. By sending messages at varying frequencies, and attaching listeners in the plugin to record the rate of datagram ingress, *PlatformOcean* is able to profile its own messaging throughput. This process is called *dogfooding* – using one’s own software to further develop (or analyse) the software itself.

This *profiler* plugin comprises two core functions. Firstly, we attach an autonomous message-generating agent to the plugin that repeatedly invokes the message-sending API to send arbitrary messages over the network. This mechanism is shown in Listing 6.1.

```

1 function useAgent({ runProfiler, MPS, sendMessage }) {
2   // manually handle infinite timeout
3   useEffect(() => {
4     if (MPS === 0 || !runProfiler) return;
5
6     const intervalMs = 1000 / MPS;
7
8     const interval = setInterval(() => {
9       const length = Math.ceil(2 + Math.random() * 5);
10      const fakeMessage = Math.random()
11        .toString(36)
12        .slice(2, 2 + length);
13      props.sendCreateMessage(fakeMessage, false);
14    }, intervalMs);
15
16    // cleanup when MPS/runProfiler changes or component unmounts
17    return () => clearInterval(interval);
18  }, [MPS, runProfiler, sendMessage]);
19 }

```

Listing 6.1: High-frequency message generating agent for *PlatformOcean* profiler

The randomly-generated message payloads are anywhere between lengths two and seven, with an expected length of five. When attaching the relevant metadata from Table 5.3, this entails disseminating 118 bytes on average per send, given that one byte is allocated per character.

Secondly, we attach a recording mechanism to measure the throughput of both message ingress, and subsequent rendering to the UI. Similarly to Listing 5.1, we attach a listener to watch for changes in the size of the message queue, and record that a message has been received when the message queue is polled. We show this mechanism in Listing 6.2.

```

1 useEffect(() => {
2   // no message events to process
3   if (messageQueueLen === 0) return;
4
5   // message is received
6   const { message } = props.pollMessageQueue();
7   // record event to MPS counter
8   recordEvent();
9   setFeed((prev) => {
10    const next = prev.length >= 10 ? prev.slice(1) : prev;
11    return [...next, message.message];
12  });
13 }, [messageQueueLen]);

```

Listing 6.2: Throughput recording mechanism for *PlatformOcean* profiler

The throughput is then recorded using a moving average filter. Each second, a new sample of the total number of received messages is taken and stored in a queue of length five, which is then averaged. The resulting plugin is shown in Figure 6.9.

We can then define the message ingress rate as the *perceived MPS*, and the *efficiency* as the

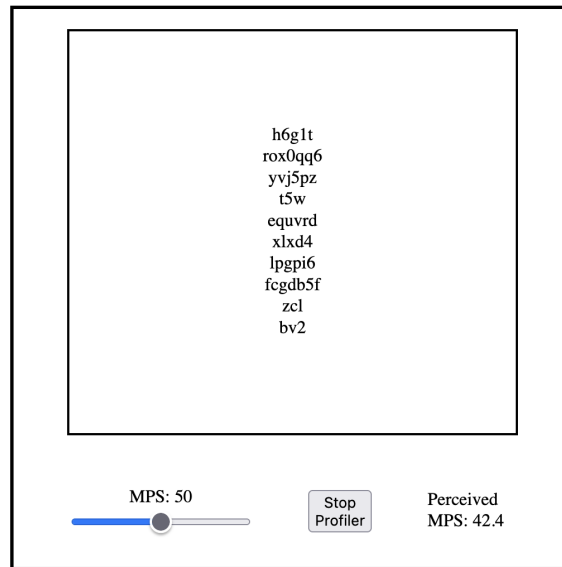


Figure 6.9: Interface of *profiler* plugin. The MPS is varied using the slider on the left-hand side and recorded in the *Perceived MPS* window

ratio of perceived messages to the true message egress rate. By varying the true MPS from 0 to 100 in increments of 5, we generate the results shown in Figure 6.10.

Figure 6.10 allows for a theoretical bound to be calculated for the maximum MPS tolerance. By extrapolating the *efficiency* LOBF we can find the x-axis intercept, which defines the theoretical saturation point of 0% efficiency. We show this calculation in Equation 6.1.

$$96.043 \div 0.259 \approx 371MPS \quad (6.1)$$

This demonstrates that *PlatformOcean* can theoretically tolerate up to 370 messages per second. Of course, this result is limited by two assumptions. Firstly, it is not possible to have truly zero efficiency, as this is only achievable with a message ingress rate of zero. In reality, the efficiency would decrease *asymptotically* towards zero, as the hard upper bound is reached for increasing egress speeds. This entails that there is no ‘true’ zero-crossing point, and the x-axis intercept should be interpreted as an estimate derived from the fitted model rather than a strict upper bound.

Secondly, the experiment utilises a *simple* user interface that consumes messages directly from the queue, with limited preprocessing. As a result, the measured throughput reflects not only the performance of the underlying messaging infrastructure, but also the overhead associated with the *React* rendering pipeline and browser execution environment. The ‘true’ messaging throughput

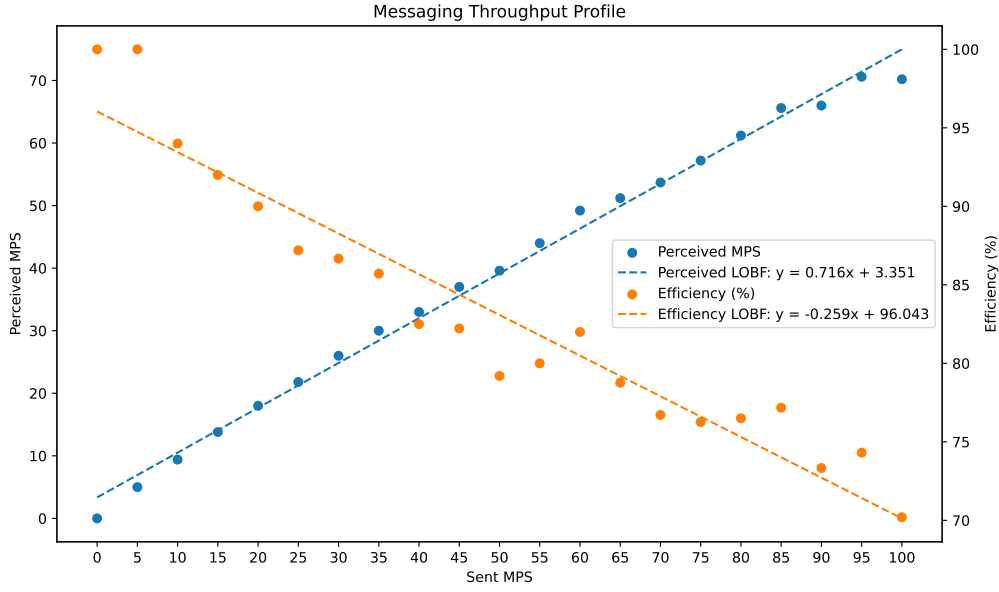


Figure 6.10: Plots for *perceived MPS* and *efficiency* for varying speeds of message egress. For each plot, the *line of best fit* (LOBF) is recorded in the legend

rate is inextricably linked to the preprocessing overheads, and therefore imposes a strict caveat that *results may vary by application*.

6.9 Summary

PlatformOcean is underpinned by its distributed, plugin-based architecture. Accordingly, platform instances spawned by *PlatformOcean* can scale in two dimensions: either in the number of stand-alone instances coalesced into an ecosystem, or the number of plugins customising each of the platform instances.

In this section, we have evaluated the platformiser across these two dimensions of plugin and platform cardinality, ranging from single platform, single plugin, to multi platform, plugin. For each instantiation, we have provided screenshots of the running platform instance, as well as an example data structure used for codifying the plugin state for dissemination using the *PlatformOcean* API.

While there is a clear motivation for the choice of testcases, this set is of course not exhaustive. A natural follow-up question might therefore be: *are there any types of systems that cannot be implemented in PlatformOcean?* Having demonstrated *completeness* in Chapter 4, the answer should (in principle) be *no!* In practice, however, just because the state can be represented through an arbitrary payload, it does not entail that hardware implementation is appropriate.

For example, a high-frequency trading would need ultra-fast, specialised algorithms, rather than a generic representation that's sufficient for 'normal' applications. Furthermore, profiling would suggest that *PlatformOcean* can tolerate $\approx 371MPS$, while *Twitter* requires $\approx 10,000MPS$ for production-grade deployment. This shortcoming can be attributed to the design philosophy of the platformiser: for the purpose of ground-up, social coordination, we promote flexibility over specialisation.

Note that this discussion of the case studies has become progressively higher-level from the social media stereotypes to the multi-plugin, multi-platform systems. This has meant that the later case studies have focussed on the presentation of the platform functionality, rather than how the concepts of Chapter 5 have been applied. However, complete details about the implementation of the different interaction contracts can be found at <https://github.com/MattSScott/PlatformOceanPlugins>.

Summary and Conclusions

Contents

7.1 Summary	139
7.2 Limitations	141
7.3 Further Work	143
7.4 Epilogue	145

7.1 Summary

The aim of this thesis was to demonstrate that an alternative exists to the ecosystem of centralised, monopolistic platforms offered by profit-driven actors who do not necessarily experience the consequences, unintended or otherwise, of their design choices and technological deployment [170]. To counteract this proliferation of mainstream platforms, we have introduced the counter-process of *ethical platformisation* (EP), and eight corresponding **EP Principles**. These principles ensure, among other things, that stakeholder values can be incorporated into the platform ecosystem not merely at design time, but *repeatedly* at run time. In order to guide this process of *ethical platformisation*, and thereby create a viable alternative to the ecosystem of platforms produced by (standard) platformisation, an appropriate tool which we termed a *platformiser* was required. This tool was offered in the form of *PlatformOcean*, one possible implementation of a *platformiser*.

The viability of this *platformiser* has been established over five steps. By considering the necessity for *ethical platformisation* to be performed and guided by a *platformiser*, we have demonstrated that this kind of tool is *arguable*.

Furthermore, we have proposed three constituent components for one such platformiser: a *platform*, providing mechanisms for distributed message passing, user registration and bespoke customisation; *plugins* remote micro-frontend modules that serve to customise and enhance the functionality of the platform; and a *meta-platform*, serving as a repository for both plugins and pre-instantiated platform instances. Having decomposed the *platformiser* in this way, we have demonstrated that it is *specifiable*.

By giving a semantics to an abstract platform representation, representing it in terms of its bounded contexts, interaction contracts, and non-functional requirements, we have constructed a mapping function to refine this abstract representation into a concrete plugin architecture, itself comprising a kernel and a set of dynamically loadable plugins. Having shown this mapping explicitly, we demonstrate that the platforms generated by this platformiser are sufficiently verbose to re-implement any existing platform, and that the *platformiser* is *platform complete*. We have therefore demonstrated that the *platformiser* is *computable*.

We have subsequently implemented each of the three constituent components of the *platformiser* explicitly, by providing the relevant algorithms and data structures for the platform, meta-platform, and plugins. We have also provided links to a working, full-stack prototype, available at <https://github.com/PlatformOcean>. Having built this *platformiser* tool, we have shown it is *implementable*.

Finally, we have applied this *platformiser* across six test cases, investigating the two dimensions of ecosystem scalability: platform cardinality, where the number of platforms in a given application ecosystem vary; and plugin cardinality, where the number of plugins used to customise a given platform instance vary. The implementation of these testcases is accessible at <https://github.com/MattSScott/PlatformOceanPlugins>. By stress-testing the *platformiser* in this way, we have shown that it is *applicable*.

Accordingly, we have demonstrated that *PlatformOcean*, one possible implementation of a *platformiser*, is five-sevenths¹ of the way to being a fully viable alternative to the centralised and monopolistic platform solutions that are currently present in the Digital Society. The remaining two issues of *adoptability* and *acceptability* are necessarily addressed in further work: verifying the viability of a dynamically changing ecosystem requires a longitudinal study, rather than a single time slice. *Adoptability* is only assessable through long-term platform usage, and engagement in the platform ecosystem – we want to observe the long-term evolution, or even *emergence*, of platforms,

¹or 71.4%, which still equates to a first-class degree

rather than base a judgement on any single instantaneous configuration. Furthermore *acceptability* must be assessed not just by metrics and relative costs, but by a system that synergises individual and collective satisfaction, e.g., interactional justice [171].

Ultimately, this thesis can be viewed through two lenses. If read *forwards*, one can see the development of this *platformiser* as a journey of revelation: going through each step iteratively builds upon previous concepts to edge closer and closer to a cohesive, testable piece of software. Alternatively however, if read *backwards*, one can see this thesis as an exercise in revealed wisdom: starting from its functional requirements and implementation, we demonstrate the steps required to build and verify any tool of this kind, in its own way.

7.2 Limitations

The proof-of-concept implementation of the *PlatformOcean* platformiser involved making several design choices, which potentially present some limitations to functionality and usability. This includes security, plugin architecture genericity, the degree of centralisation, and plugin misalignment. This section discusses each of these limitations in turn.

Firstly, the plugin architecture underpinning the distributed client-server model creates a trade-off between high platform uptime and plugin genericity, and security. In particular, the use of *Webpack* and its *Dynamic Remotes* enables users to specify generic, versatile plugins (so long as they conform to the *PlatformOcean* API). These plugins are subsequently bundled and minified, thereby encoding and compressing them into *JSX*-compatible code for dynamic injection into a platform. This process is termed *remote code execution*, and carries security risks. Due to the highly-generic plugin SDK, it becomes difficult to ensure that plugin developers do not engineer *Byzantine* (i.e., intentionally malicious) plugins. While *PlatformOcean* platforms to implement rudimentary sandboxing through *iframes*, and the SDK imposes guardrails on the data pathways, there are still mechanisms for getting around this; malicious developers can still exploit the opacity of the minification system to hide malicious code.

Beyond the possible security issues, the genericity of the plugin architecture also limits their capacity for extension. At the platform level, plugins are dynamically registered and given a unique identifier. This identifier is only known at runtime (i.e., at time of injection), rather than compile time (i.e., during the implementation of the plugin). Client-to-client cross communication is handled by the platform through both its wrapper component and API: the unique plugin identifier

is propagated to the client, which makes the appropriate subscriptions, and then attaches the networking methods to the plugin via the wrapper API. In this way, 1-to-1 plugin communication is trivial, as developers simply use the methods defined by the wrapper to handle state synchronisation. This creates a limitation for plugin multiplexing, however. In order to access datagrams from other plugins, thereby enabling plugin-to-plugin cross-communication, the topic subscription route must be known. Since this route is generated at runtime on the platform, a plugin developer cannot know about it at design time. We provide a proof-of-concept solution to this problem through the intra- and inter-plugin subscription routes in the wrapper API: developers can use a separate information pathway to listen for datagrams from an external plugin. This therefore enables a 1-to- N plugin-to-plugin communication system: N plugins can listen to a single ‘source’ plugin by using the inter-communication protocol. This protocol has limitations for an M -to- N communication system, however, as developers would have to manually filter by communication pathway and perform manual multiplexing.

7

Furthermore, while *PlatformOcean* makes the architectural choice to implement its components as a distributed client-server model, there is still overlap with the *centralised* client-server model. In particular, just like in a centralised model, the *meta-platform* and *plugins* both manifest on the cloud; it is only the *platform* ecosystem that instead manifests as a distributed, locally-owned server. This design choice redresses many of the concerns of power asymmetries and algorithmic misinformation, since platforms are engineered ground-up by communities, rather than top-down from the software providers, but can still introduce concerns with ownership of the *meta-platform*. This concern is addressed in part through the open-source software implementation and arms'-length relationships between provider and user, but would be better addressed *technically* through a clearer separation in architectures – for example a *peer-to-peer* implementation. This would also help produce a more stable, long-term *meta-platform*, rather than the temporary solution offered by a free CDN.

Finally, the introduction of a separate, command-line SDK introduces opportunities for misalignment between plugins in local development mode and plugins in distributed production mode. In order to avoid plugin developers having to instantiate an entire server for plugin development, we create a local mock-up: state management occurs through local *React* primitives (e.g., *setState* in a parent component) rather than true distributed message operations (i.e., network requests between client and server). There is therefore a key trade-off between streamlining the development process and ensuring a complete alignment between plugins in development and plugins in production. This alignment can be minimised by ensuring closer parallels between the different

transport mechanisms. Although it is unfeasible for plugin developers to instantiate an entire ‘dev’ server, the routing can better mirror a ‘true’ distributed system by implementing the same serialise, transport, deserialise pattern with artificial delays in *setTimeout*, rather than the guaranteed instantaneous transmission through *setState*.

7.3 Further Work

In moving from a prototype to a production-grade system, it may be necessary to revisit the inter-plugin protocol to permit bi-directional dataflow, similar to a *WebSocket*. This would enable cross-purpose datagrams, where plugin *A* can send contracts to (and enforce changes upon) plugin *B*, and *vice versa*. This may entail that, beyond just storing plugin hashes, the meta-platform *also* stores the contract grammars defined on the plugin. Prospective federated plugins can then pre-emptively know to emit, or accept, grammars of that type.

Such a two-way approach could enable scenarios like SMP supporting not just one, but a set of alternative plugins that can source the nutrient information in different ways (assuming they all support the same interaction contract). SMP could then be wired up by a platform provider to use one data source or another.

One central contribution of *PlatformOcean* is its capacity for *generativity* – i.e., the capacity for the platforms offered by the platformiser to evolve and transcend the initial vision of the provider. In order for the platforms to evolve, however, they must go through several iterations. New communities must build on and redevelop previous platform solutions by augmenting and changing their plugin configuration. Much like evolution, this process is not instantaneous. What is therefore required is a *longitudinal* study with full *stakeholder engagement*, where a single platform concept is studied across multiple user groups, showing how these different communities want to use their platforms from the start, but also how the plugin configurations change over time.

In addition to performing a longitudinal study of the *PlatformOcean* platformiser solution, it would be beneficial to develop two complementary, generic plugins through the SDK. One would be for *governance*, and the other would be for *transactions*. Firstly, the governance plugin would need to recognise the trade-off between community expertise, problem complexity, and tool configuration [172]: for example, majority voting is likely to be satisfactory for simple problems of the same type but unsuitable in long-lived communities requiring compromises and the repeated solving of wicked problems that don’t tend to have a guaranteed termination state. Secondly, by recognising

that transactions in goods and services are a fundamental feature of human co-existence, we also want a generic plugin that supports of principles of *axial currencies* [173], for example using blockchain technologies. This plugin would disincentivise currency monopolisation and instead promote regular transactions.

A natural extension to, and trajectory of, the *PlatformOcean* ecosystem is to introduce opportunities for *federation*. This would entail that discrete platform instances have the ability to, at the datagram-transport level, interoperate to enable platform *A* to send and receive datagrams from platform *B*. In a sense, this targets the same mechanism as plugin federation, albeit at a larger scale; just as federated plugins must be aligned at the contract level, so too should federated platforms be aligned at the plugin level. This approach differs, however, since plugin-to-plugin brokerage occurs *on* the platform, whereas platform-to-platform brokerage must occur *between* platforms. This, naturally, requires an extension to the current set of platform protocols, such that, similarly to how plugins may currently be linked together, so too can platforms be linked together to form message distribution chains.

However, federation is a complex subject, and can manifest in multiple different ways. The *Fediverse* considers a protocol-driven approach whereby independent social media and web publishing servers can communicate with each other [68]. Alternatively, *grassroots federation* considers coalescing platform sub-systems into broader, hierarchical super-systems [69], thereby creating platforms of platforms. In the *PlatformOcean* paradigm, it is not unreasonable to implement a ‘socio-political’ (i.e., non-technological) kind of federation, where members from a variety of different platforms use another, dedicated platform as a common substrate. This naturally raises the question as to what kind of authority is required for making the federation: complexity arises through the need to respect Ostrom’s 7th and 8th design principles for institutions [58]. Consequently, further work requires a substantive analysis of platform federation, both in terms of its parameters, and examples of instantiation.

Finally, we aim to leverage the platform ecosystem to produce a dedicated platform to serve as a communication substrate for plugin developers and platform users. In the same way that self-hosting systems like *GitHub* documented and version controlled their source code using *GitHub*, companies exhibit *dogfooding* (eating one’s own ‘dog food’) by intentionally using their own platform internally, and social media companies have their own accounts, so too can we implement a *dogfooded PlatformOcean* discussion platform using *PlatformOcean*. Using this mechanism is the best way to foster platform evolution, and strengthen coherence between developer and user experience.

7.4 Epilogue

PlatformOcean is more than a standalone system for personal use. It is a system that has been tried and tested by others, albeit so far in undergraduate projects. For example, as a prototype implementation of an end-to-end platformiser, the *Mind My Mind* platform was implemented by *Hana Khalil* for her Final Year M.Eng. project. Similarly, an *Axial Currency* plugin has been developed by *Morgan Pressman* in his Final Year M.Eng. project. After initial induction and training, both students were able to independently use the platformiser, and to produce (i.e., *platformise*) robust platforms with substantive functionality. (An instance of *Mind My Mind* was even observed being used by other undergraduate students.)

In addition to the external use of the platformiser by others, *PlatformOcean* has been presented and published at the *International Symposium on Technology and Society* (ISTAS) [109]. This research gave original insight both *abstractly*, into the potential social impact of ethical platformisation, and *concretely*, through the full-stack architecture. Three further submissions have been made (under review at the time of submission), on a deep conceptualisation of ethical platformisation the formal proof of platform completeness, and how the final system architecture constitutes a novel intervention into the techno-politics of platformisation.

Field testing of the platformiser ‘in the wild’ has begun with *stakeholder engagement* at the 11th *African Day of School Feeding* (11th *ADSF*), an international conference focusing on integrating nutritious meals with water, sanitation, and hygiene (WASH). This conference, in particular, operated under the theme *Ensuring Access to Nutritious Meals*, and provided opportunities for meeting and collaborating with representatives from a variety of African countries. Through collaboration with the *School of Public Health* at *Imperial*, we demonstrated the *CSSMA* application from Section 6.7 as a *sustainable meal planning* (SMP) tool to a range of prospective stakeholders.

Through interacting with the various representatives, a consistent pair of features kept emerging for how such a tool should look. This tool should be *decentralised*, thereby re-empowering the local African stakeholders, and *customisable*, such that various stakeholders can engage with the tool in different ways. Consequently, *PlatformOcean* was very well received at the conference, and was requested to be included in a broader SMP toolkit. Following from the 11th *ADSF*, we have engaged in co-production with six African countries, and given an interactive demonstration of a mock-up version of *CSSMA* at a regional workshop on *21st Century School Meals in Africa*, subtitled *Unlocking Benefits for Human and Planetary Health*.

Finally, *PlatformOcean* can be seen as more than just an example of a platformiser, but also as an instance of a design pattern. Decomposing software components into a *meta-platform* for instantiating *platforms* that are themselves composed by *plugins* mirrors the *hierarchical modularity* software pattern [174]. In this way, a one-to-one correspondence exists between the meta-platform and libraries, the platform and packages, and the plugins and modules. The meta-platform provides a library, which itself provides multiple packages. Each platform then serves as a package which can be customised using plugins, or modules. Leveraging the experience gained from developing *PlatformOcean* according to this design pattern, we used it to also implement *basePlatformSOMAS* – a generic, modular library that provides a *baseServer*, *baseAgent*, and *baseMessage* package, all of which can be extended and swapped in and out to produce a fully functional multi-agent system for different applications.

basePlatformSOMAS has been used to implement the Assessed Coursework component of the Self-Organising Multi-Agent Systems course offered in the Electrical and Electronic Engineering Department at Imperial College London [175]. In particular, it has been applied in the context of using multi-agent simulation to investigate self-organisation in co-operative survival ‘games’ [176]; for example it was used in a computational simulation of an algorithmic model of Terror Management Theory [177]. In particular, though, *basePlatformSOMAS* has been used to implement the *Megabike* scenario. *Megabike* considers a grid world of multi-rider bicycles, each bike crewed by eight agents. Independently, these agents can steer and pedal, with their actions aggregated together to manoeuvre the bike. Movement actions cost energy, which can be replenished by acquiring lootboxes. This general framework has been used to investigate the efficiency differences between social deliberation and social contracts [178], an information-theoretic analysis of different leadership systems [179], and the emergence of social cohesion under different governance regimes [180]. As well as *PlatformOcean*, *basePlatformSOMAS* is also available on GitHub at github.com/MattSScott/basePlatformSOMAS.

In summary, while this thesis marks a complete description of a research project, it also marks a waypoint in a research programme, with multiple educational, theoretical and software engineering directions to pursue.

A

A

Appendix: Supplementary Algorithms

```

1  const aggregate_dicts = (dict_a, dict_b) => {
2    let agg_dict = {}
3    for (const key of Object.keys(dict_a)) {
4      // ensure compatability
5      if (!(key in dict_b)) {
6        throw new Error("Objects have different shape")
7      }
8      let a_val = dict_a[key]
9      let b_val = dict_b[key]
10     if (a_val === null || b_val == null) {
11       throw new Error(`Found null value with key: ${key}`)
12     }
13     let a_val_type = typeof a_val
14     let b_val_type = typeof b_val
15     if (a_val_type !== b_val_type) {
16       throw new Error(`Found type mismatch with key: ${key}`)
17     }
18
19     // aggregate dicts
20     if (a_val_type == "number") {
21       agg_dict[key] = a_val + b_val
22     }
23     else if (a_val_type == "object") {
24       let agg_sub = aggregate_dicts(a_val, b_val)
25       agg_dict[key] = agg_sub
26     }
27     else {
28       throw new Error(`Unsupported dictionary key: found ${a_val_type}, want: "number
29         " or "object"`)
30     }
31   }
32   return agg_dict
33 }

```

Listing A.1: Object aggregator function for CSSMA tool. Given two JavaScript objects (dictionaries) with M and N keys, respectively, this algorithm aggregates the two dictionaries (i.e., performs a pairwise sum across keys) with $O(M + N)$ time complexity in the worst case. This function leverages the self-similarity in the representation of foods, meals, and menus, as a nested dictionary of nutritional information. This therefore provides a low complexity, linear function for a frequently-called procedure, and terminates because each recursive call reduces the problem to either a (nullable) single value or smaller object

Algorithm 4 Recursive constraint search for valid meals in the *CSSMA* tool. Given N ingredients, and a maximally-satisfiable constraint M to optimise for, the worst-case time complexity is $O(M^N)$, as M of each ingredient can be chosen. In reality, the branching factor (and therefore search space) is relatively small, since the numerical data per ingredient is of a similar order of magnitude to M . Similarly, N is constrained due to a relatively small ingredient pool. This algorithm ensures all possible solutions to a given set of constraints and terminates since it adapts the standard 0/1 knapsack into a vectorial knapsack

Require: O – vector of ingredients

Require: r – vector of nutrient requirements

Require: x – vector of partial ingredient solution

Ensure: X (array of valid solutions)

```

1:  $n \leftarrow \text{card}(O)$ 
2:  $i \leftarrow 0$ 
3: function SEARCH( $i, r, x$ )
4:   if  $r \leq 0$  then
5:      $X \leftarrow X \cup \{x\}$ 
6:     return
7:   end if
8:   if  $i > n$  then
9:     return
10:  end if
11:  for  $k = 0$  to  $\maxUse(O_i, r)$  do
12:    SEARCH( $i + 1, r - k \cdot O_i, x \cup \{x_i = k\}$ )
13:  end for
14: end function

```

Bibliography

- [1] J.-C. Rochet and J. Tirole, “Platform competition in two-sided markets,” *Journal of the european economic association*, vol. 1, no. 4, pp. 990–1029, 2003.
- [2] C. Y. Baldwin, C. J. Woodard, et al., “The architecture of platforms: A unified view,” *Platforms, markets and innovation*, vol. 32, pp. 19–44, 2009.
- [3] M. De Reuver, C. Sørensen, and R. C. Basole, “The digital platform: A research agenda,” *Journal of information technology*, vol. 33, no. 2, pp. 124–135, 2018.
- [4] A. Tiwana, *Platform ecosystems: Aligning architecture, governance, and strategy*. Newnes, 2013.
- [5] J.-C. Plantin, C. Lagoze, P. N. Edwards, and C. Sandvig, “Infrastructure studies meet platform studies in the age of google and facebook,” *New media & society*, vol. 20, no. 1, pp. 293–310, 2018.
- [6] S. Zuboff, “Surveillance capitalism and the challenge of collective action,” in *New labor forum*, Sage Publications Sage CA: Los Angeles, CA, vol. 28, 2019, pp. 10–29.
- [7] N. D. Schüll, “Addiction by design: Machine gambling in las vegas,” in *Addiction by design*, Princeton university press, 2025.
- [8] A. S. George, A. George, T. Baskar, and M. Karthikeyan, “Reclaiming our minds: Mitigating the negative impacts of excessive doomscrolling,” *Partners Universal Multidisciplinary Research Journal*, vol. 1, no. 3, pp. 17–39, 2024.
- [9] IEEE, “IEEE standard model process for addressing ethical concerns during system design,” *IEEE Std 7000-2021*, pp. 1–82, 2021. DOI: 10.1109/IEEESTD.2021.9536679.
- [10] Signal, <https://signal.org>, 2026.
- [11] A. Nowak and R. Vallacher, “Nonlinear societal change: The perspective of dynamical systems,” *British Journal of Social Psychology*, vol. 58, no. 1, pp. 105–128, 2019.
- [12] R. Bartle, “Hearts, clubs, diamonds, spades: Players who suit muds,” *Journal of MUD research*, vol. 1, no. 1, p. 19, 1996.

-
- [13] T. Poell, D. Nieborg, and J. Van Dijck, "Platformisation," *Internet Policy Review*, vol. 8, no. 4, pp. 1–13, 2019.
 - [14] D. S. Evans, "Some empirical aspects of multi-sided platform industries," *Review of Network Economics (RNE)*, vol. 2, no. 3, 2003.
 - [15] R. Kane, *What is a two-sided marketplace*, <https://zapier.com/blog/two-sided-marketplace/>, 2024.
 - [16] K. B. Clark, "The interaction of design hierarchies and market concepts in technological evolution," *Research policy*, vol. 14, no. 5, pp. 235–251, 1985.
 - [17] Y. Yoo, O. Henfridsson, and K. Lyytinen, "Research commentary—the new organizing logic of digital innovation: An agenda for information systems research," *Information systems research*, vol. 21, no. 4, pp. 724–735, 2010.
 - [18] D. Kellner, "Globalization, technopolitics and revolution," in *Technology and democracy: toward a critical theory of digital technologies, technopolitics, and technocapitalism*, Springer, 2021, pp. 153–177.
 - [19] J. Kallinikos, A. Aaltonen, and A. Marton, "The ambivalent ontology of digital artifacts," *MIS quarterly*, pp. 357–370, 2013.
 - [20] R. Wolfinger, D. Dhungana, H. Prähofer, and H. Mössenböck, "A component plug-in architecture for the .net platform," in *7th Joint Modular Languages Conference*, Springer, 2006, pp. 287–305.
 - [21] R. Oldenburg, *The great good place: Cafes, coffee shops, bookstores, bars, hair salons, and other hangouts at the heart of a community*. Da Capo Press, 1999.
 - [22] R. Oldenburg, "Our vanishing third places," *Planning commissioners journal*, vol. 25, no. 4, pp. 6–10, 1997.
 - [23] J. Finlay, M. Esposito, M. H. Kim, I. Gomez-Lopez, and P. Clarke, "Closure of 'third places'? exploring potential consequences for collective health and wellbeing," *Health & place*, vol. 60, p. 102225, 2019.
 - [24] J. Kim, T. Klein-Balajee, R. M. Kelly, and A. Hiniker, *Discord's design encourages "third place" social media experiences*, 2025. arXiv: 2501.09951 [cs.HC]. [Online]. Available: <https://arxiv.org/abs/2501.09951>.
 - [25] B. Thornton, "The emergence of digital third place & the waning role of physical third place," M.S. thesis, Pittsburg State University, Jul. 2024. DOI: 10.13140/RG.2.2.14599.61605.

-
- [26] S. Graham and S. Marvin, *Splintering urbanism: networked infrastructures, technological mobilities and the urban condition*. Routledge, 2002.
- [27] A. Oswal, <https://www.paloaltonetworks.com/blog/2024/06/platformization-unified-platforms>, 2024.
- [28] R. Abbas, J. Pitt, and K. Michael, “Socio-technical design for public interest technology,” *IEEE Transactions on Technology and Society*, vol. 2, no. 2, pp. 55–61, 2021.
- [29] R. Abbas, S. Hamdoun, J. Abu-Ghazaleh, N. Chhetri, N. Chhetri, and K. Michael, “Co-designing the future with public interest technology,” *IEEE Technol. Soc. Mag.*, vol. 40, no. 3, pp. 10–15, 2021.
- [30] N. Bontridder and Y. Pouillet, “The role of artificial intelligence in disinformation,” *Data & Policy*, 3:e32, 2021.
- [31] S. Zuboff, “The age of surveillance capitalism,” in *Social theory re-wired*, Routledge, 2023, pp. 203–213.
- [32] J. Williams, *Stand out of our light: Freedom and resistance in the attention economy*. Cambridge University Press, 2018.
- [33] L. Parisi, “Reprogramming decisionism,” *E-Flux*, vol. 85, 2017.
- [34] A. Anikina, “Automating platform spectators: Algorithmic montage and affective scroll in tiktok,” *Digital Culture and Society*, vol. 7, no. 2, pp. 117–136, 2022.
- [35] K. Pohl, *Requirements engineering: An overview*. RWTH, Fachgruppe Informatik Aachen, 1996.
- [36] L. A. Macaulay, *Requirements engineering*. Springer Science & Business Media, 2012.
- [37] D. Graeber and D. Wengrow, *The Dawn of Everything*. London, UK: Penguin Books, 2021.
- [38] G. I. Webb, R. Hyde, H. Cao, H. L. Nguyen, and F. Petitjean, “Characterizing concept drift,” *Data Mining and Knowledge Discovery*, vol. 30, no. 4, pp. 964–994, 2016.
- [39] J. Lu, A. Liu, F. Dong, F. Gu, J. Gama, and G. Zhang, “Learning under concept drift: A review,” *IEEE transactions on knowledge and data engineering*, vol. 31, no. 12, pp. 2346–2363, 2018.
- [40] A. Vasalou, A. Joinson, T. Bänziger, P. Goldie, and J. Pitt, “Avatars in social media: Balancing accuracy, playfulness and embodied messages,” *International Journal of Human-Computer Studies*, vol. 66, no. 11, pp. 801–811, 2008.

-
- [41] J. Pitt, "From trust and loyalty to lock-in and digital dependence," *IEEE Technology and Society Magazine*, vol. 39, no. 1, pp. 5–8, 2020.
 - [42] R. Johnson and A. Cureton, "Kant's moral philosophy," *The Stanford Encyclopedia of Philosophy*, 2004.
 - [43] O. O'Neill, *Acting on principle: An essay on Kantian ethics*. Cambridge University Press, 2013.
 - [44] J. Rawls, "Kantian constructivism in moral theory," *The journal of philosophy*, vol. 77, no. 9, pp. 515–572, 1980.
 - [45] J. Pitt, A. Mertzani, M. Scott, and C. Smit, "The architecture of re-empowerment," *IEEE Technology and Society Magazine*, vol. 44, no. 1, pp. 74–86, 2025.
 - [46] J. Pitt, A. Mertzani, and J. Ober, "Self-governing systems," *Frontiers in Political Science*, vol. 7, 2025.
 - [47] J. Ober, *Demopolis: Democracy before liberalism in theory and practice*. Cambridge: Cambridge University Press, 2017.
 - [48] J. Pitt and J. Ober, "Democracy by design: Basic democracy and the self-organisation of collective governance," in *2018 IEEE 12th International Conference on Self-Adaptive and Self-Organizing Systems (SASO)*, 2018, pp. 20–29.
 - [49] J. Pitt, J. H. Clippinger, and C. Sorensen, "Values, axial currencies, and computational axiology: Digital currencies can do more than buy stuff," *IEEE Technology and Society Magazine*, vol. 37, no. 3, pp. 56–63, 2018.
 - [50] B. Malinowski, "Kula; the circulating exchange of valuables in the archipelagoes of eastern new guinea," *Man*, vol. 20, pp. 97–105, 1920.
 - [51] A. Mertzani, J. Pitt, A. Nowak, and T. Michalak, "Expertise, social influence, and knowledge aggregation in distributed information processing," *Artificial Life*, vol. 29, no. 1, pp. 37–65, 2023.
 - [52] P. E. Petruzzi, J. Pitt, and D. Busquets, "Electronic social capital for self-organising multi-agent systems," *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, vol. 12, no. 3, pp. 1–25, 2017.
 - [53] A. Helmond, "The platformization of the web: Making web data platform ready," *Social media+ society*, vol. 1, no. 2, p. 2056305115603080, 2015.
 - [54] S. Zuboff, "Big other: Surveillance capitalism and the prospects of an information civilization," *Journal of information technology*, vol. 30, no. 1, pp. 75–89, 2015.

-
- [55] Y. Varoufakis, *Technofeudalism: What killed capitalism*. Melville House, 2024.
- [56] A. Rychwalska, M. Roszczyńska-Kurasińska, K. Ziembowicz, and J. V. Pitt, “Fitness for purpose in online communities: Community complexity framework for diagnosis and design of socio-technical systems,” *Frontiers in Psychology*, vol. 12, p. 739415, 2021.
- [57] J. L. Zittrain, “The generative internet,” *Harv. L. Rev.*, vol. 119, p. 1975, 2005.
- [58] E. Ostrom, *Governing the Commons*. Cambridge, UK: Cambridge University Press, 1990.
- [59] R. Michels, *Political parties: A sociological study of the oligarchical tendencies of modern democracy*. Dover, 1959.
- [60] A. Tiwana, B. Konsynski, and A. Bush, “Platform evolution: Coevolution of platform architecture, governance, and environmental dynamics (research commentary),” *Information Systems Research*, vol. 21, no. 4, pp. 675–687, 2010.
- [61] P. A. Laplante and M. Kassab, *Requirements engineering for software and systems*. Auerbach Publications, 2022.
- [62] M. Gien, “Micro-kernel architecture key to modern operating systems design,” *Unix review*, vol. 8, no. 11, pp. 58–60, 1990.
- [63] R. Thomas, “Microkernel architecture,” *Lecture Notes in Software Engineering*, 2025.
- [64] S. Schaupp, “Technopolitics from below: A framework for the analysis of digital politics of production,” *NanoEthics*, vol. 15, no. 1, pp. 71–86, 2021.
- [65] L. Lessig, “Code is law,” *Harvard magazine*, vol. 1, no. 2000, 2000.
- [66] L. Lessig, *Code: And other laws of cyberspace*. ReadHowYouWant. com, 2009.
- [67] W. J. Mitchell, *City of bits: space, place, and the infobahn*. MIT press, 1996.
- [68] L. La Cava, S. Greco, and A. Tagarelli, “Understanding the growth of the fediverse through the lens of mastodon,” *Applied network science*, vol. 6, pp. 1–35, 2021.
- [69] E. Shapiro, *A grassroots architecture to supplant global digital platforms by a global digital democracy*, 2024. arXiv: 2404.13468 [cs.NI]. [Online]. Available: <https://arxiv.org/abs/2404.13468>.
- [70] K. Kapoor, A. Z. Bigdeli, Y. K. Dwivedi, A. Schroeder, A. Beltagui, and T. Baines, “A socio-technical view of platform ecosystems: Systematic review and research agenda,” *Journal of Business Research*, vol. 128, pp. 94–108, 2021.
- [71] J. Van Dijck, T. Poell, and M. De Waal, *The platform society: Public values in a connective world*. Oxford university press, 2018.

-
- [72] M. G. M. Nyabuto, V. Mony, and S. Mbugua, "Architectural review of client-server models," *International journal of scientific research and engineering trends*, vol. 10, no. 1, pp. 139–143, 2024.
 - [73] A. Berson, *Client/server architecture*. McGraw-Hill, Inc., 1992.
 - [74] C. S. Guynes and J. Windsor, "Revisiting client/server computing," *Journal of Business & Economics Research (JBER)*, vol. 9, no. 1, 2011.
 - [75] C. Cimen, Y. Kavurucu, and H. Aydin, "Usage of thin-client/server architecture in computer aided education.," *Turkish Online Journal of Educational Technology-TOJET*, vol. 13, no. 2, pp. 181–185, 2014.
 - [76] R. M. Adler, "Distributed coordination models for client/server computing," *Computer*, vol. 28, no. 4, pp. 14–22, 1995.
 - [77] J. Ober, *Demopolis: Democracy before liberalism in theory and practice*. Cambridge University Press, 2017.
 - [78] J. H. Saltzer, D. P. Reed, and D. D. Clark, "End-to-end arguments in system design," *ACM Transactions on Computer Systems (TOCS)*, vol. 2, no. 4, pp. 277–288, 1984.
 - [79] J. Wu, *Distributed system design*. CRC press, 2017.
 - [80] S. Pitt, M. van Meelis Lacey, E. Scaife, and J. Pitt, "No app is an island: Collective action and sustainable development goal-sensitive design," *IJIMAI*, vol. 6, no. 5, pp. 24–33, 2021.
 - [81] J. Shorish, "Blockchain state machine representation," *Open Science Framework* (<https://osf.io/hbmje/>), 2018.
 - [82] D. Burkhardt, M. Werling, and H. Lasi, "Distributed ledger," in *2018 IEEE international conference on engineering, technology and innovation (ICE/ITMC)*, IEEE, 2018, pp. 1–9.
 - [83] R. Herian, "The politics of blockchain," *Law and Critique*, vol. 29, no. 2, pp. 129–131, 2018.
 - [84] M. Swan, *Blockchain: Blueprint for a new economy*. " O'Reilly Media, Inc.", 2015.
 - [85] E. Deirmentzoglou, G. Papakyriakopoulos, and C. Patsakis, "A survey on long-range attacks for proof of stake protocols," *IEEE access*, vol. 7, pp. 28 712–28 725, 2019.
 - [86] V. Sus, "Proof-of-stake is a defective mechanism," *Cryptology ePrint Archive*, 2022.
 - [87] Y. Shifferaw and S. Lemma, "Limitations of proof of stake algorithm in blockchain: A review," *Zede Journal*, vol. 39, no. 1, pp. 81–95, 2021.

-
- [88] A. Gervais, G. O. Karame, K. Wüst, V. Glykantzis, H. Ritzdorf, and S. Capkun, “On the security and performance of proof of work blockchains,” in *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*, 2016, pp. 3–16.
 - [89] M. Platt et al., “The energy footprint of blockchain consensus mechanisms beyond proof-of-work,” in *2021 IEEE 21st International Conference on Software Quality, Reliability and Security Companion (QRS-C)*, IEEE, 2021, pp. 1135–1144.
 - [90] X. Yang, Y. Chen, and X. Chen, “Effective scheme against 51% attack on proof-of-work blockchain with history weighted information,” in *2019 IEEE International Conference on Blockchain (Blockchain)*, IEEE, 2019, pp. 261–265.
 - [91] E. Shapiro, “Grassroots systems: Concept, examples, implementation and applications,” *arXiv preprint arXiv:2301.04391*, 2023.
 - [92] F. Wang and P. De Filippi, “Self-sovereign identity in a globalized world: Credentials-based identity systems as a driver for economic inclusion,” *Frontiers in Blockchain*, vol. 2, p. 28, 2020.
 - [93] W. Kerber and H. Schweitzer, “Interoperability in the digital economy,” *J. Intell. Prop. Info. Tech. & Elec. Com. L.*, vol. 8, p. 39, 2017.
 - [94] L. De Lauretis, “From monolithic architecture to microservices architecture,” in *2019 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, IEEE, 2019, pp. 93–96.
 - [95] R. Salaverriá, M.-P. Martínez-Costa, and C. González Tosat, “Decentralised networks as a tool for fighting disinformation and censorship: The fediverse and free, collaborative and open networks,” in *Journalism, Digital Media and the Fourth Industrial Revolution*, Springer, 2024, pp. 15–25.
 - [96] D. Birsan, “On plug-ins and extensible architectures: Extensible application architectures such as eclipse offer many advantages, but one must be careful to avoid “plug-in hell.”,” *Queue*, vol. 3, no. 2, pp. 40–46, 2005.
 - [97] J. Männistö, A.-P. Tuovinen, and M. Raatikainen, “Experiences on a frameworkless micro-frontend architecture in a small organization,” in *2023 IEEE 20th International Conference on Software Architecture Companion (ICSA-C)*, IEEE, 2023, pp. 61–67.
 - [98] M. L. Katz and C. Shapiro, “Network externalities, competition, and compatibility,” *The American economic review*, vol. 75, no. 3, pp. 424–440, 1985.

-
- [99] C. Shapiro and H. R. Varian, *Information rules: A strategic guide to the network economy*. Harvard Business Press, 1999.
 - [100] L. Fridman, *Transcript for pavel durov: Telegram, freedom, censorship, money, power & human nature*, <https://lexfridman.com/pavel-durov-transcript>, 2025.
 - [101] A. Preukschat and D. Reed, *Self-sovereign identity*. Manning Publications, 2021.
 - [102] T. Hardjono, P. Deegan, and J. H. Clippinger, “Social use cases for the id3 open mustard seed platform,” *IEEE Technology and Society Magazine*, vol. 33, no. 3, pp. 48–54, 2014.
 - [103] S. Zuboff, “Big Other: Surveillance capitalism and the prospects of an information civilization,” *Journal of Information Technology*, vol. 30, pp. 75–89, 2015.
 - [104] N. Srnicek, *Platform Capitalism*. Cambridge, UK: Polity Press, 2016.
 - [105] J. Pitt, A. Mertzani, M. Scott, and C. Smit, “The architecture of re-empowerment,” *IEEE Technol. Soc. Mag.*, vol. 44, no. 1, pp. 74–86, 2025.
 - [106] C. Are, “An autoethnography of automated powerlessness: Lacking platform affordances in instagram and tiktok account deletions,” *Media, Culture & Society*, vol. 45, no. 4, pp. 822–840, 2023.
 - [107] C. Are, “Researching under the platform gaze: Rethinking the challenges of platform governance research,” *Platforms & Society*, vol. 1, 2024.
 - [108] E. Shapiro and N. Talmon, *Grassroots federation: Fair governance of large-scale, decentralized, sovereign digital communities*, 2025. arXiv: 2505.02208 [cs.DC]. [Online]. Available: <https://arxiv.org/abs/2505.02208>.
 - [109] M. Scott, C. Smit, and J. Pitt, “A system architecture for ethical platformisation,” in *2023 IEEE International Symposium on Technology and Society (ISTAS)*, IEEE, 2023, pp. 1–10.
 - [110] M. Scott and J. Pitt, *A full-stack platform architecture for self-organised social coordination*, 2025. arXiv: 2507.01239 [cs.NI]. [Online]. Available: <https://arxiv.org/abs/2507.01239>.
 - [111] J. Pitt, A. Rychwalska, M. Roszczynska-Kurasinska, and A. Nowak, “Democratizing platforms for social coordination,” *IEEE Technol. Soc. Mag.*, vol. 38, no. 1, pp. 43–50, 2019.
 - [112] M. Abadi and L. Lamport, “The existence of refinement mappings,” *Theoretical Computer Science*, vol. 82, no. 2, pp. 253–284, 1991.
 - [113] E. Prodromou, *ActivityPub: Programming for the Social Web*. ” O’Reilly Media, Inc.”, 2024.

-
- [114] S. Pitt, M. Lacey, E. Scaife, and J. Pitt, “No app is an island: Collective action and sustainable development goal-sensitive design,” *International Journal of Interactive Multimedia and Artificial Intelligence*, vol. 6, pp. 24–33, Mar. 2021. DOI: 10.9781/ijimai.2021.02.005.
- [115] J. Derrick, E. Boiten, and S. Reeves, “Proceedings 15th international refinement workshop,” *arXiv preprint arXiv:1106.3488*, 2011.
- [116] D. Marmosier and G. Blakqori, “Apml: An architecture proof modeling language,” in *International Symposium on Formal Methods*, Springer, 2019, pp. 611–630.
- [117] R. A. Riemenschneider, “Checking the correctness of architectural transformation steps via proof-carrying architectures,” in *Working Conference on Software Architecture*, Springer, 1999, pp. 65–81.
- [118] F. Honsell, J. Longley, D. Sannella, and A. Tarlecki, “Constructive data refinement in typed lambda calculus,” in *International Conference on Foundations of Software Science and Computation Structures*, Springer, 2000, pp. 161–176.
- [119] M. Hazewinkel, “Encyclopaedia of mathematics: Fibonacci method,” in Dordrecht: Springer Netherlands, 1989, ch. H, pp. 468–469. DOI: 10.1007/978-94-009-5997-2_3.
- [120] B. Rossman, “Homomorphism preservation theorems,” *Journal of the ACM (JACM)*, vol. 55, no. 3, pp. 1–53, 2008.
- [121] E. Evans, *Domain-driven design: tackling complexity in the heart of software*. Addison-Wesley Professional, 2004.
- [122] O. Özkan, Ö. Babur, and M. van den Brand, “Domain-driven design in software development: A systematic literature review on implementation, challenges, and effectiveness,” *Journal of Systems and Software*, p. 112 537, 2025.
- [123] DEV, *Domain driven design*, <https://dev.to/lovestaco/domain-driven-design-3i2j>, 2025.
- [124] Microsoft, *Common web application architectures*, <https://learn.microsoft.com/en-us/dotnet/architecture/modern-web-apps-azure/common-web-application-architectures>, 2023.
- [125] M. H. Meyer and A. P. Lehnerd, *The power of product platforms*. Simon and Schuster, 1997.
- [126] V. Krishnan and S. Gupta, “Appropriateness and impact of platform-based product development,” *Management science*, vol. 47, no. 1, pp. 52–68, 2001.
- [127] M. G. Jacobides, C. Cennamo, and A. Gawer, “Externalities and complementarities in platforms and ecosystems: From structural solutions to endogenous failures,” *Research Policy*, vol. 53, no. 1, p. 104 906, 2024.

-
- [128] S. Kapferer and O. Zimmermann, “Domain-specific language and tools for strategic domain-driven design, context mapping and bounded context modeling.,” in *MODELSWARD*, 2020, pp. 299–306.
 - [129] S. Kapferer and O. Zimmermann, “Domain-driven architecture modeling and rapid prototyping with context mapper,” in *International Conference on Model-Driven Engineering and Software Development*, Springer, 2020, pp. 250–272.
 - [130] Baeldung, *DDD bounded contexts and java modules*, <https://www.baeldung.com/java-modules-ddd-bounded-contexts>, 2024.
 - [131] A. A. Júnior, S. Misra, and M. S. Soares, “A systematic mapping study on software architectures description based on iso/iec/ieee 42010: 2011,” in *International Conference on Computational Science and Its Applications*, Springer, 2019, pp. 17–30.
 - [132] M. Haoues, A. Sellami, H. Ben-Abdallah, and L. Cheikhi, “A guideline for software architecture selection based on ISO 25010 quality related characteristics,” *International Journal of System Assurance Engineering and Management*, vol. 8, no. Suppl 2, pp. 886–909, 2017.
 - [133] ISO Architecture, *Defining architecture*, IEEE 1471 / ISO/IEC 42010 working group website, Accessed September 5, 2025, 2011. [Online]. Available: <http://www.iso-architecture.org/ieee-1471/defining-architecture.html>.
 - [134] D. L. Parnas, “On the criteria to be used in decomposing systems into modules,” *Communications of the ACM*, vol. 15, no. 12, pp. 1053–1058, 1972.
 - [135] M. Tsechelidis, N. Nikolaidis, T. Maikantis, and A. Ampatzoglou, “Modular monoliths the way to standardization,” in *Proceedings of the 3rd Eclipse Security, AI, Architecture and Modelling Conference on Cloud to Edge Continuum*, 2023, pp. 49–52.
 - [136] R. Su and X. Li, “Modular monolith: Is this the trend in software architecture?” In *Proceedings of the 1st International Workshop on New Trends in Software Architecture*, 2024, pp. 10–13.
 - [137] N. Gonçalves, D. Faustino, A. R. Silva, and M. Portela, “Monolith modularization towards microservices: Refactoring and performance trade-offs,” in *2021 IEEE 18th International Conference on Software Architecture Companion (ICSA-C)*, IEEE, 2021, pp. 1–8.
 - [138] T. Rentsch, “Object oriented programming,” *ACM Sigplan Notices*, vol. 17, no. 9, pp. 51–57, 1982.
 - [139] W. R. Ashby, “Requisite variety and its implications for the control of complex systems,” in *Facets of systems science*, Springer, 1991, pp. 405–417.

-
- [140] L. Bass, *Software architecture in practice*. Pearson Education India, 2012.
 - [141] Medium, *Design twitter*, <https://pratikraj.medium.com/design-twitter-4e4be1da84c1>, 2022.
 - [142] Medium, *Whatsapp at scale: A guide to non-functional requirements*, <https://levelup.git-connected.com/whatsapp-nfrs-2acd855073c3>, 2025.
 - [143] S. Frey and R. Sumner, “Emergence of integrated institutions in a large population of self-governing communities,” *PloS one*, vol. 14, no. 7, e0216335, 2019.
 - [144] K. Zateishchikov, “Scaling a software platform using micro frontends,” Ph.D. dissertation, VAMK, 2023.
 - [145] C. Lee and S. Helal, “Protocols for service discovery in dynamic and mobile networks,” *International Journal of Computer Research*, vol. 11, no. 1, pp. 1–12, 2002.
 - [146] C. Bettstetter, C. Renner, et al., “A comparison of service discovery protocols and implementation of the service location protocol,” in *Proceedings of the 6th EUNICE Open European Summer School: Innovative Internet Applications*, 2000.
 - [147] Signal, *SignalService*, <https://github.com/signalapp/libsignal-service-java/blob/master/protobuf/SignalService.proto>, 2019.
 - [148] D. Dossot, *RabbitMQ essentials*. Packt Publishing Ltd, 2014.
 - [149] D. Betts, J. Dominguez, G. Melnik, F. Simonazzi, and M. Subramanian, *Exploring CQRS and Event Sourcing: A journey into high scalability, availability, and maintainability with Windows Azure*. Microsoft patterns & practices, 2013.
 - [150] M. Fowler, *Event sourcing*, <https://martinfowler.com/eaDev/EventSourcing.html>, 2005.
 - [151] Redis, *Redis pub/sub*, <https://redis.io/docs/latest/develop/pubsub/>, 2026.
 - [152] P. Noordhuis, *Simple demo to showcase redis pubsub with eventmachine*, <https://gist.github.com/pietern/348262>, 2010.
 - [153] A. Alves, *OSGI in Depth*. Simon and Schuster, 2011.
 - [154] OWASP, *Html5 security cheat sheet*, https://cheatsheetseries.owasp.org/cheatsheets/HTML5_Security_Cheat_Sheet.html#sandboxed-frames, 2026.
 - [155] P. D. Manuel and J. AlGhamdi, “A data-centric design for n-tier architecture,” *Information Sciences*, vol. 150, no. 3-4, pp. 195–206, 2003.
 - [156] Z. Jackson, *Module federation examples*, <https://github.com/module-federation/module-federation-examples/tree/master/advanced-api/dynamic-remotes>, 2023.

-
- [157] Y. Zheng and X. Zhang, "Path sensitive static analysis of web applications for remote code execution vulnerability detection," in *2013 35th International Conference on Software Engineering (ICSE)*, 2013, pp. 652–661.
- [158] A. Szabo and U. Hadad, *Crypto miner attack: Gpu remote code execution attacks*, 2025. arXiv: 2502.10439 [cs.CR]. [Online]. Available: <https://arxiv.org/abs/2502.10439>.
- [159] F. Lin, H. Wang, L. Wang, and X. Liu, "A longitudinal study of removed apps in ios app store," in *Proceedings of the Web Conference 2021*, 2021, pp. 1435–1446.
- [160] R. Bodle, "Regimes of sharing: Open apis, interoperability, and facebook," *Information, Communication & Society*, vol. 14, no. 3, pp. 320–337, 2011.
- [161] A. Rockley and P. Kostur, *Managing enterprise content: A unified content strategy*. New Riders, 2003.
- [162] Drupalize, *What are plugins?* <https://drupalize.me/tutorial/what-are-plugins>, 2022.
- [163] L. Wiederkehr, J. Pitt, T. Dannhauser, and K. Bruzda, "Attention enhancing technology: A new dimension in the design of effective wellbeing apps," *IEEE Transactions on Technology and Society*, vol. 2, no. 3, pp. 157–166, 2021.
- [164] B. Alderson-Day and C. Fernyhough, "Inner speech: Development, cognitive functions, phenomenology, and neurobiology," *Psychological bulletin*, vol. 141, no. 5, p. 931, 2015.
- [165] W. F. Programme, <https://publications.wfp.org/2022/state-of-school-feeding/>, 2022.
- [166] D. A. Bundy, *Rethinking school feeding: social safety nets, child development, and the education sector*. world bank publications, 2009.
- [167] S. Pastorino et al., "Planet-friendly school meals: Opportunities to improve children's health and leverage change in food systems," *The Lancet Planetary Health*, vol. 9, no. 11, 2025.
- [168] D. A. Bundy et al., "Investment in child and adolescent health and development: Key messages from disease control priorities," *The Lancet*, vol. 391, no. 10121, pp. 687–699, 2018.
- [169] L. Drake, A. Woolnough, C. Burbano, and D. A. Bundy, *Global school feeding sourcebook: lessons from 14 countries*. world scientific, 2016.
- [170] K. Michael, "Modern indentured servitude in the gig economy: A case study on the deregulation of the taxi industry in the united states," *IEEE Technology and Society Magazine*, vol. 41, no. 2, pp. 30–41, 2022.

-
- [171] J. Pitt, "Interactional justice and self-governance of open self-organising systems," in *2017 IEEE 11th International Conference on Self-Adaptive and Self-Organizing Systems (SASO)*, IEEE, 2017, pp. 31–40.
- [172] A. Rychwalska, M. Roszczyńska-Kurasińska, K. Ziembowicz, and J. V. Pitt, "Fitness for purpose in online communities: Community complexity framework for diagnosis and design of socio-technical systems," *Frontiers in Psychology*, vol. 12, p. 739415, 2021.
- [173] J. Pitt, J. H. Clippinger, and C. Sørensen, "Values, axial currencies, and computational axiology: Digital currencies can do more than buy stuff," *IEEE Technol. Soc. Mag.*, vol. 37, no. 3, pp. 56–63, 2018.
- [174] M. Blume and A. W. Appel, "Hierarchical modularity," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 21, no. 4, pp. 813–847, 1999.
- [175] J. Pitt, M. Scott, M. Suvarayan, A. Dimoska, A. Mertzani, and C. Smit, "Software engineering and society engineering for self-organising multi-agent systems," in *Go Where the Bugs Are: Essays Dedicated to Wolfgang Reif on the Occasion of His 65th Birthday*, Springer, 2025, pp. 104–131.
- [176] M. Scott and J. Pitt, "Interdependent self-organizing mechanisms for cooperative survival," *Artificial Life*, vol. 29, no. 2, pp. 198–234, 2023.
- [177] J. Pitt, M. Scott, and A. Shah, "A matter of a (life and death): A computational model of terror management theory," in *Artificial Life Conference Proceedings 37*, vol. 2025, 2025, p. 56.
- [178] M. Scott, A. Mertzani, C. Smit, S. Sarkadi, and J. Pitt, "Social deliberation vs. social contracts in self-governing voluntary organisations," in *International Workshop on Coordination, Organizations, Institutions, Norms, and Ethics for Governance of Multi-Agent Systems*, Springer, 2024, pp. 57–75.
- [179] M. Scott, M. Sas, and J. Pitt, "An information-theoretic analysis of leadership in self-organised collective action," in *2024 IEEE International Conference on Autonomic Computing and Self-Organizing Systems (ACSOS)*, IEEE, 2024, pp. 101–110.
- [180] M. Stevenson, M. Scott, M. Sas, and J. Pitt, *An information-theoretic analysis of the emergence of social cohesion*, ALife Workshop on Agent-Based Modelling og Human Behavior (ABMHuB), 2025. [Online]. Available: http://abmhub.cs.ucl.ac.uk/2025/camera_ready/Stevenson_etal.pdf.