philosophies

MDPI

Article

The Machine in the Ghost: Autonomy, Hyperconnectivity, and Residual Causality

Barry M. O'Reilly

Department of Complexity and Design, School of Engineering and Innovation, The Open University, Milton Keynes MK7 6AA, UK; barry@blacktulip.se

Abstract: This article will examine the unnamed and potentially devastating constraining effect of software on human autonomy. I call this concept residual causality, where software design decisions made long ago in different circumstances for different reasons constrain human action in an unknown future. The less aware the designers of software systems are of complexity in social systems, the more likely they are to introduce residual causality. The introduction of intricate, ordered machines, to a world largely defined by disorder and heuristics, has caused philosophical perturbations that we have not fully dealt with. The *machine in the ghost* is the belief that machine thinking can be applied to the environment in which the machine will operate. As hyperconnectivity increases, the ghost becomes more unpredictable, unmanageable, and even less like the machine. If we continue to indulge the machine view of the world, the design of software systems presents real dangers to the autonomy of the individual and the functioning of our societies. The steadfastness of machine ontologies in the philosophies of software architects risks creating increasing residual causality as hyperconnectivity increases. Shifting the philosophical position of software architects opens up the possibility of discovering new methods that make it easier to avoid these dangers.

Keywords: residuality; antifragility; complexity; hyperliminal systems; hyperliminal coupling

Citation: O'Reilly, B.M. The Machine in the Ghost: Autonomy, Hyperconnectivity, and Residual Causality. *Philosophies* **2021**, *6*, 81. https://doi.org/10.3390/philosophies6040081

Academic Editor: Marcin J. Schroeder

Received: 30 July 2021
Accepted: 27 September 2021
Published: 30 September 2021

1. Introduction

Hyperconnectivity is something that has touched every aspect of modern life: "it has transformed social interaction, culture, economics, politics, and the self." [1]. Hyperconnectivity introduces dependencies and risks for cascading errors that could potentially have wide ranging consequences, described as extraordinary risk [2]. This article considers how naively engineered software may contribute to that risk.

In June 2021, a ransomware attack on the COOP chain of grocery stores in Sweden [3] closed 800 stores across the country. The entire payment process for consumers was dependent on the single software component targeted by the attack. Whilst attention will focus on how to avoid this happening again, the more interesting question is how a single design decision could bring such an enormous system to a halt. Millions of people along multiple supply chains had contributed to the consumer experience inside the stores—and a single design decision was enough to disrupt this, resulting in enormous financial consequences.

The COOP incident is not unique. As hyperconnectivity increases, the design decisions of software engineers have the potential to cause cascading, unseen problems. A modern city is expected to experience an increasing integration between technology and citizens. Billions of sensors will direct decision making, AI software agents will make bureaucratic decisions, medical software will make automated diagnoses, transport systems will be automated by algorithm, and social media systems will provide communication and dissemination of information. The connections and dependencies between these technological advancements and between the societal environment emerge invisibly as we deploy them. Software architects are unaware of the unintended consequences of their designs in the

context of the hyperconnected society. These unintended consequences are called residual causality [4].

Unexpected events, such as accidents, flooding, fires, occupation, terrorist attacks, or something as unthreatening as an election, or normal fluctuations in the business cycle, may suddenly reveal the unseen interdependencies across these systems. Home security systems, weather warning systems, medical devices, and automated cars could be thrown into unseen patterns of behavior just when they are needed most. Recent years have also highlighted how hyperconnectivity has led to amplified divisions, inflaming conflict and making societal structures vulnerable. Our hyperconnected society is a deeply entangled network of dependencies between people, organizations, and technologies too intricate and dynamic to map. We are becoming ever more reliant on this network, and its connections, relationships, intricacies and communications are dependent on software built using the same decision making tools and frameworks as those used at COOP and everywhere else in the industry.

The COVID-19 crisis has shown how vulnerable our supply chains are [5], how quickly disinformation and panic can spread, and how difficult it can be when the expectation of order is not met. As the pandemic nears its end, a new world will emerge. It will involve disrupted supply chains being reconfigured, knowledge workers being reluctant to travel, and demand fluctuations as we reassess what is important to us. This will translate inevitably to a huge backlog of software changes at large companies. It will be first at the point of design of these changes that we will see the impacts of residual causality. The design decisions behind older software systems, built for a pre COVID world, may well slow our progress to a new normal, and may even cause unforeseen difficulties of their own. This has yet to happen, but similarities exist with Brexit, with concern that software changes would be enormous and difficult [6]. Astoundingly there is no academic literature on this situation for either Brexit or COVID. This lack of awareness of the problem is described in later sections. Issues similar to the COOP problem will continue to be a surprise every time it happens.

More sinisterly, the ability of residual causality to control and restrict human options in the event of stress will not go unnoticed by those who actively seek to control or disrupt. The ripple effects of these software design decisions could be leveraged to create problems, and even become attack vectors against us.

Science fiction has described nightmare scenarios where malevolent, self-aware software systems attack human society. Very few have imagined a future where the inability of software to respond to its environment, poor quality, can present the biggest risk to human autonomy.

As hyperconnectivity increases, it will be necessary for software developers to be aware of these kinds of risks and to meaningfully engage with trying to prevent them. This presents a number of challenges to the designers of software systems. In hyperconnected systems, standard approaches to risk identification and requirements elicitation [7], become subject to non-ergodicity [8], where prediction of the future events or future needs in the system simply is not possible. This challenges software engineers to find a new basis on which to make decisions in this kind of environment. This kind of environment is very different than that of the universe imagined by mathematicians, and as such the tools of software engineering reside within an entirely different paradigmatic basis than the environment where software engineering is being carried out. The assumption of this article is that residual causality can be reduced by reducing this mismatch.

This article identifies this theoretical gap in the software industry. Furthermore, it makes the claim that in order to fill this gap, software engineering needs to become more aware of the philosophical traditions underlying its practice, and be prepared to shift these traditions to uncover new methods of developing software. The review of popular software engineering strands of thinking shows that most approaches are trapped in a reductionist, Newtonian way of seeing the world, described as the component metaphor [4], and this restricts the ability of software designers to think in new ways. It is considered in this

article that this way of thinking ensures that current methods of software engineering cannot keep up with ever increasing hyperconnectivity and are unprepared to deal with the risks presented.

The hypothesis is that philosophical investigation of the beliefs of software engineers will produce new ways of developing software for a world that is constantly changing. The article will discuss one particular approach that has been developed in this way, residuality theory, and the implications of this for the future of software engineering.

2. Why Software Is Different: Hyperliminality

When we make software design decisions we couple components to each other. This coupling introduces dependencies between those components, and this opens the door for contagion. Contagion occurs when an issue affects a particular component, all those components which are coupled to it are also affected. This opens the door for cascading errors, poorly understood ripples that can have effects far beyond what we can imagine.

Software engineering has for decades focused on the coupling between these components. However, there has been less focus on how they are also coupled to their environment, such as societal ideas, economic fluctuations, political movements, and competitor behaviors. The lack of academic articles investigating the impacts of Brexit on software structures, despite the urgency felt by industry and institutions, highlights this gap [6].

Software systems cannot therefore be approached as standalone entities with clear boundaries like mechanical systems such as cars or aeroplanes. They are considered *hyperliminal* [4]; with an ordered, highly constrained technical component in the software and a disordered component in the organization, market, and society in which the software will execute. In this hyperliminal environment, coupling can exist between software components that is impossible for the designer to see at the outset of a project. This coupling is related to the disordered environment, which at some point in the future will make itself known through stress. I have named this invisible, unpredictable coupling *hyperliminal coupling* [9].

The software that we need to navigate problems on a global scale will suffer from two things—the ignorance of hyperliminal coupling and the rigidity of component based design. These two things combine to limit our possible responses under stress or change—this is residual causality [4].

In response to these issues, modern software engineering has abandoned the traditional systems engineering that saw planning and specification as key. Hyperliminal coupling reveals itself so quickly that often software cannot be built according to a plan or specification because the intended dependencies and relationships between components and the environment are frequently knocked out of kilter before work is completed.. The software industry ascribed these problems to the speed of change, and sought methodologies, loosely grouped together as *agile*, that focused on responding quickly to change whilst keeping current design methodologies intact. The idea that we will be able to adapt does not allow for the fact that residual causality is the thing that decides which avenues we have for adaptation: that the paths forward can be extremely limited by earlier decision making, and that the component metaphor is one of the most limiting factors. For these paths forward, in every component metaphor decision, the designer of the system discounts a huge number of future potential situations simply because the designer cannot imagine them. Standard methods of developing software introduce the very problems that the software industry suffers from: an inability to build the right system for the environment that we expect it to operate in [7]. Thus current engineering techniques are not built to work in hyperconnected environments, and new thinking will be necessary.

3. Software's Philosophical Basis: The Component Metaphor

The hypothesis contests that the current ontological and epistemological positions of software architects are too limited to cope with a hyperconnected world where uncertainty

is increasing. In this section, the current philosophical position of software engineers is discussed.

In the aftermath of the industrial revolution, it was easy to feel a sense of pride in the achievements of enlightenment thinking, our mastery of the world around us. Every motor car, with more than 30,000 separate parts, a collaboration of more than 10,000 people, that rolled of the production line was a triumph over the complexity of our world. Scaling and improving this process increased the sense of mastery.

Whilst the miracle of the modern car factory is to be lauded, using the lessons of the industrial revolution to frame and develop the processes for software development results in a *component metaphor* [4]. Here, the development of software is assumed to involve the hunt for an optimum component breakdown. This breakdown is based on the understanding of the organization or task as a number of fixed processes or use cases, often seeing the organization using the software as a simple, mechanistic machine itself.

There is no literature on the philosophical position of the designers of software systems, and as such, no data. This at once suggests a huge gap in the literature. However, analyzing the popular literature paints a picture of the position of today's software engineers. The philosophy of software architecture can then be said to be accidental, inherited from its mathematical roots and the question largely ignored [4]. Most academic articles within computer science do not mention the paradigmatic basis of the researchers, since a straight forward post-positivist approach is assumed. (However, articles where aspects of the component metaphor are projected onto non software systems, such as organizational design, are easy to find.).

Uncertainty in the Literature

Trying to understand the philosophical position of software architects in relation to hyperconnectivity puts us at the intersection of three fields: the complexity sciences, systems engineering, and software engineering. Whilst uncertainty is described across the literature, most evidently in the complexity sciences and surrounding fields of systems sciences [10–13], it seems to be less present in systems engineering and appears even less present in the software engineering literature. It has long been argued that systems engineering cannot answer the problems of complexity [14,15], although systems engineering provides excellent tools for managing the software itself. When uncertainty is acknowledged, the solutions are often described in terms of anticipation [16–18], the ability of the architect to somehow sense the future. Taleb [11] presents a different way of dealing with uncertainty, less about defining probabilities and more about survival, vulnerability, and awareness of the limitations of prediction. Similar ideas from the field of safety are reflected in works by Woods [17,18] and more relevant to software engineering by Hole [19]. Hole however does not describe how these software systems were designed, merely that they do exist. Much of the software literature also emphasizes the ability to anticipate unknown futures through scenario analysis techniques like ATAM [20], experimentation [21] or through outright prediction [22]. Uncertainty in the academic literature often falls back on statistical techniques like Bayesian methods [23] and appears to have an underlying assumption that eventually we will predict the future. Methods for reducing uncertainty include sensemaking, anticipation [24], hazard analysis [16], classical risk management, and in software, the analysis of language, either through requirements analysis or modeling of stakeholder's language [25,26].

The 2018 paper by Patriarca [27] details the gap in models that allow systems engineering approaches to model uncertainty, and it is this gap that this residuality theory attempts to address.

The field of software engineering has since the late 60's been concerned with boundary decisions in software as the means to manage uncertainty in hyperliminal environments. By setting appropriate component boundaries, software engineers believed that software could be made more flexible and thus respond easier to changes in the environment. Many methodologies prescribe pattern approaches such as Fowler [28] or Gang of Four [29] as

a way to solve this problem. Other approaches, such as Domain Driven Design [25] and requirements engineering [26] focus on language and clarity in order to improve quality. Mnemonics like SOLID [30], GRASP [31], and DRY [32] provide hints as to how boundaries should be set in any project, independent of context and represent pattern approaches.

However, despite this body of popular work being widely referenced and known, most software developers will report that component boundary decisions are made based on gut feeling, and experience, or some idea of what a best practice is [33]. Even when trained in these methodologies the vast majority of software developers and architects appear unsure of how to make boundary decisions [33].

> "*while code appears full of decisions, coders are often not conscious of their choices or alternatives*" (Ralph, 2016) [33].

One approach is simple prediction. The IDesign Method [22] uses the concept of volatility based decomposition, leaning on the earlier work of Parnas [34] called non-conventional decomposition. This, at least, provides some basis for decision making in software engineering, but requires that the architect is able to predict what will change in the future.

The limitations of software engineering, systems engineering, and the complexity sciences mean that there is currently no recorded way to design software systems for conditions of uncertainty. Current solutions describe how resilient or extensible systems might look, but very little is new since the 1960s early exploration of modularity in software engineering. Outside of Taleb's work on uncertainty, there is nothing to suggest that this will be met anytime soon. This situation is considered entirely acceptable because the future is considered to be unknowable, and thus there is nothing to be done.

4. A Possible Solution: Residuality Theory

Residuality theory [35] may solve the problem of the elicitation of requirements and the assessment of risk in uncertain environments. These methods indulge in a constant reductionism, making the model of the environment smaller in order to easier make software decisions. These reduced models become a Baudrillardian simulacra, and attention is then focused on the model and the real world ignored. Residuality theory pushes back in the opposite direction, exploring rather than reducing. It represents a huge shift and transforms the "deep V" of systems engineering, allowing component design to emerge as a result of the application of stress, rather than designing first and then applying stress.

Residuality theory states that the future of a complex system is a function of its residues. A residue is what is left over when a stressor impacts the system. A residue usually consists of software functions, infrastructure, people, and information flows between them. It is a deliberately open concept. In real world modelling it contains anything relevant to the environment and the particular stressor. Residuality theory is concerned with design decisions in hyperliminal systems. Residuality theory models the hyperliminal system as a stack of interconnected residues. It applies random stressors during design to identify residues, and to investigate a system's ability to withstand unknown sources of stress. This broadens the scope of the design effort far beyond the immediate functional concerns of sponsors and engineers. Most modeling techniques require the software architect to already begin describing the use cases and solution in order to model. This causes all kinds of confirmation biases to influence the design process. Residuality uses novel techniques, such as playfulness in the stressor description, and requires no assessment of probability or impact before a stressor is introduced. It borrows ideas from machine learning such as bagging and boosting and training/testing sets. Instead of mitigating risks one at a time, or identifying components immediately, residuality theory seeks instead to identify common states, called attractors, in the hyperliminal environment, and find those that show the greatest tendencies to withstand unknown forms of stress. Thus, residuality theory implies a methodology which is an *attractor search*, rather than requirements elicitation or risk management.

There are three major ideas that support the use of residuality theory in software systems:

(1) Exposure to stress is what makes systems stronger and more likely to survive unknown stressors [11].
(2) Design in a hyperliminal environment should be performed as an attractor search instead of the identification of individual requirements and risks, as these are forms of reduction and prediction in an unpredictable, and therefore irreducible, space.
(3) The number of points of decoupling that an architect can impact in a software system is significantly less than the number of potential stressors. Since many of the coupling points are invisible, the application of longer lists of stressors reveals more hyperliminal coupling.

Residual analysis is the process created by applying residuality theory to software engineering. It consists of a stressor analysis and a contagion analysis. The stressor analysis transforms the list of random stressors to a set of residues. The contagion analysis transforms the set of residues to a set of coupling decisions, which we call the architecture of the software system. By applying random sources of stress in the stressor analysis, the designer is constantly forced to make changes to boundary decisions, redundancy, and diversity. The set of stressors is made as large as possible by removing the probability/impact filters usually employed in the component metaphor. By using one set of stressors to train the architecture and arrive at an emerging component structure, a second set can be used to test this architecture and confirm that the system shows signs of surviving unknown forms of stress. Bagging and boosting means that we can reorder the stressor list and perform this multiple times, each time finding attractors that will appear to perform better or worse than others, and uncovering an enormous amount of hyperliminal coupling that would otherwise remain unseen through traditional risk management approaches, or through the component metaphor's focus on requirements and functionality. Unlike other approaches, this training/testing set technique provides an empirical means by which to show that the design effort has been effective in producing more resilient designs.

Stuart Kauffman [36] showed via simulators that complex systems made up of individual, autonomous elements tended to arrive at a point of stability and equilibrium over time. The interesting point was that these systems arrived at a much smaller number of equilibrium states than that which was available to them. These states are described as *attractors*—points of equilibrium to which the system repeatedly returns. Kauffman interpreted this as a possible law, that order would emerge as movement toward a smaller set of possibilities than that which actually existed in the system. If these attractors exist in hyperliminal systems, then residuality theory provides a systematic means for uncovering them. If they do not exist, then the design effort is destined to be incredibly turbulent and residuality theory confers no disadvantages over the component metaphor.

Software systems also have a finite number of points of internal coupling which the architect can actually impact. The number of potential stressors is infinite, and therefore a random walk through potential stressors, free from the reductionist biases of enterprise risk management, has a greater chance of revealing hidden hyperliminal coupling that can then be addressed by the architecture.

By modeling a system as a series of interconnected residues, engineers can express and investigate the behavior of a system outside immediate functional concerns, and thus be better placed to make decisions about the interaction of the system and its environment. Instead of trying to manage hazards and identify risks, as other methodologies do, such as the Architecture Trade-off Analysis Method (ATAM) [20], System Theoretic Process Analysis (STPA) [37], or Failure Mode Effects Analysis (FMEA) [38], residuality causes the architect to perform a conscious search for attractors. System architectures that display antifragile or resilient tendencies can be separated from those that are obviously fragile in a systematic way more akin to the workings of engineering than the mysterious 'anticipation' or intuition that other methods rely on. These often vague techniques rely on our ability to know the future and are often a stubborn reiteration of the belief in order and causality [13].

Residuality allows engineers to leverage imprecision, rather than precision, to make design decisions. This is a huge step forward for engineering in uncertainty. Removing the reductionist filter of precision allows more exploratory work to be carried out. This work with stressors and contagion eventually reveals component structures that are related to the uncertainty in the environment, rather than a limited, prediction-based model of reality that component decisions are based upon in the component metaphor.

Whilst concepts such as resilience and antifragility describe the behavior of natural systems, they are less useful to engineers who have a limited scope, time pressures, and no access to true evolutionary patterns. This causes system design to suffer from the projection of the designer's beliefs in certainty onto the uncertain environment. Residuality is a better way to describe the tolerances for the unknown of hyperliminal systems, because it is directly linked to the decisions a software architect can make. Resilience and antifragility are therefore too aspirational and vague to apply to software engineering. Residuality is instead a concrete aspect of any hyperliminal system and can be measured, changed, and tested.

5. Reflexivity: Creating a New Software Engineering Theory with a Random Walk

Residuality theory is currently part of ongoing research to prove its ability to produce software systems of higher quality that are more likely to survive unpredictable conditions of stress. Anecdotally the application of the theory creates systems more likely to survive these stressors, and at the very least raises the quality of software architecture work. What is interesting is how the theory came to be created. Residuality theory is different from other ideas within software engineering, in that it is based on a pragmatic approach very far from the usual discourse within software or systems engineering. Without a push into completely different areas of thinking, starting with the complexity sciences and ending in the study of different ontologies and epistemologies, residuality theory could never have been created or articulated.

With formal training in mathematics, computer science, and software engineering, and a decades-long career designing software systems, I never once was exposed to any paradigm outside of logical positivism. Most of the influences in my career simply reiterated the component metaphor. As I began to research why it was seemingly easier for some architects to continuously deliver quality software, it became apparent that successful architects had an easier time with uncertainty, were more comfortable with missing details, and better able to plan for multiple futures. It became apparent that I and they were using techniques beyond the component metaphor, but we all struggled to articulate exactly what we were doing. Just as described by Ralph [33], both successful and unsuccessful architects struggled to explain exactly how they were making boundary decisions.

I began to understand the problem as being a limitation in the field of software engineering in dealing with uncertainty. The expectation of order in the component metaphor is caused by human understanding being thrown into flux by the introduction of the machine. The distance between the order of the machine age and the experience of everyday life was a shock, and led to a search for how to bring this kind of order to everything. This expectation, *the machine in the ghost*, is the thing that has driven scientific and engineering frustration, and led to the willful ignoring of hyperliminal coupling and the refusal to abandon the component metaphor. In order to create residuality theory, it was necessary to seek a view of the world that was somehow different.

Heidegger warned about the impact of technology [39]; that it could obscure our understanding of the natural and hinder our ability to learn, cutting us off from our environment. The spasms caused by the efficiency of the machines, their quick returns, caused us to see the world through the lens of technology, and not through other lenses. This obscures our understanding of our world and our ability to build software for it, and the second order impact of residual causality is to obscure our paths to action and understanding even more. A belief in an ordered universe akin to that of a machine

prevents the designer of software systems from looking beyond, from listening to other perspectives, and excludes the role of chance.

Recognizing that the component metaphor may be that which is holding us back, I began to investigate other ways of looking at the world. At this stage the work of Nassim Taleb was highly influential, bringing an accessible way to understand uncertainty and the kind of ideas that could be used to deal with threats and events that could not be foreseen. The Cynefin framework [12] helped to present ideas about complexity quickly and easily to other architects as I began to look outward and describe my ideas. The ideas of Ralph Stacey [13] were fundamental in building a basic understanding of causality, complexity, and Stacey's dominant discourse was the foundation for the explicit description of the component metaphor.

Each of these readings was accidental and unplanned, a random walk through the literature based on recommendations. As I progressed toward an academic investigation of these ideas, I began to understand the need to formally begin investigations within ontology and epistemology.

This led to a different set of literature being embraced. CS Pierce in 1891 defined the idea of Tychism [40]—in which much of the workings of the universe are based on chance. Pierce's most important insight, however, was that a truly disordered universe must necessarily have some aspects that are ordered—because the total avoidance of order would require a form of order to enforce this. This sits well with Prigogine's description of 'order floating on disorder' [41] and provided the basis for the concept of hyperliminal systems. Ordered systems floating on disorder is a good way to introduce the idea of hyperliminality and the connections between these as hyperliminal coupling. By making clear that the component metaphor can only apply in an ordered system, that disordered systems are subject to absolute chance, and cannot be predicted, controlled, or intentionally constrained by management until they become ordered, this ontology prevents the designers of software systems from reverting to the component metaphor. Pierce saw evolution as the combination of absolute chance and habit-taking. By accepting this absolute chance, the ghost which we have mistaken for a machine, we may increase the chances of habit taking that will allow us to build systems that are appropriate for the environments we expect them to live in. Exposure to the ideas of Tychism and Prigogine made it possible to express the idea of hyperliminality and thus have a model of reality that reconciled the uncertainty seen in software engineering with the mathematical structures in computer science.

Further reading led to a wider scope of investigation. Perhaps the attitude of Michel Serres, never completely embracing enlightenment ideas about the superiority of science, seeing a flow between the old and the new, the stories and the poems and the sciences, hold some weight here. The gestalt switch [42] is the very essence of residuality theory. The gestalt switch involves switching of background and foreground, the focus on fluctuation, noise, and randomness. Looking to the disordered, difficult part of the hyperliminal system rather than the easy, provable, predictable, ordered part. Serres states "When sciences add variety to the world, they are to be used, when they subtract variety they are to be rejected" [42]. The component metaphor is a concrete example of the reduction of variety. The ideas of Latour [43] also help to shed some light on how the component metaphor has come to be adopted, as black boxes, accepted uncritically, and passed down from each generation of software engineers as received wisdom. Here, we employ ideas on many ways of knowing from constructivism, and also employ post-structural thinking to question and deconstruct the ideas in the component metaphor. In doing so, I see every software project as a pragmatic, mixed methods adventure—a very different approach than the component metaphor, with its roots firmly in positivism.

Pierce, Heidegger, and Serres all saw the problem of the machine in the ghost, restricting and limiting our worldview, with the pretense of order at the root of the problem. Nassim Taleb describes the same problems in a much more accessible way in 'Fooled By Randomness' [10] and Ralph Stacey describes the dominant discourse of management as

working in the same vein [13]. Exposure to the same idea over and over again gave me the confidence to begin questioning the component metaphor.

Baudrillard [44] describes society as becoming detached from the natural world and reality becoming blurred by a meaningless search for complete knowledge. In a hyperconnected world, this search happens so fast that the constant stream of models makes it hard to engage with reality and instead the model becomes a simulacra, a replacement for reality. Software architecture has chosen to work with this simulacra—the component metaphor—rather than the messy, underlying reality. In this world symbolic acts such as the agile manifesto are given more meaning than they deserve. Software becomes vulnerable to sudden trends with little evidence that they actually work, and the model based detachment from reality makes it hard to argue against them. In order to avoid becoming yet another part of the model, residuality theory has avoided clear definitions of residues, encouraging exploration, questioning, and deconstruction of ideas.

This incomplete, shaky ontology, provides one platform from which software engineers can begin to extract the machine from the ghost, reconnecting with the world outside the machine in a way that is not reduced by the thinking of the component metaphor. It is from these readings that residuality theory emerged and was clarified, and similar approaches may eventually give rise to the next set of theories that give software engineering more options to try and manage the challenges of building infrastructure for the hyperconnected society.

The exploration represents an awkward stumble through the field of philosophy, something to be expected from a software engineer, but it is important not to present this as a difficult academic undertaking, or software architects will simply retract from the idea. Actual research training may be another path to explore, training architects in the basic ideas of paradigms and evidence.

Without this exploration of ideas, far outside of the realms of software engineering, it would have been impossible for me to describe what it was that I was doing differently from other architects. It is possible that many other architects are currently solving problems in a way that they cannot articulate, due to the component metaphor, after all at least 17% of enterprise software projects appear to be succeeding [45].

I noticed that there was a big difference where I chose to look at new ideas (new to me, anyway) through the lens of old ideas, it was much more difficult to frame findings and ideas from the complexity sciences in a way that was useful. Observing old ideas through the lens of new thinking was key to arriving at conclusions that allowed me to generalize enough to produce residuality theory. Instead of asking "Where are the requirements in this new model?' It was better to ask "What do requirements mean in this new model? Are they still useful, or are they getting in the way?".For a software engineer, used to working with the idea of ideation, requirements, specification, design, and testing the idea of starting with stress is like reverse engineering the process. The idea that stressors might not even exist in reality or be at all probable is difficult to reconcile with the position of efficient cause, ergodicity, or predictability. None of these positions are easy to take if the architect has never been exposed to anything except determinism.

In the literature on the philosophy of computing, no one considers the philosophy of the people making the decisions about how the software should be structured. As a practicing architect, already using some of the techniques described in residuality theory over a long and successful career, it was impossible for me to describe to other architects how I was reaching these decisions because we were all trapped in the component metaphor. By reading and widening the perspectives on the world, outside of that which an education in mathematics, computer science and systems engineering had given me, I am able to explain and articulate, and even improve the architectural process in a way that embraces uncertainty and aids in reducing residual causality.

Residuality theory suggests randomly stressing designs leads to better decision making in hyperliminal environments. In this article, the suggestion is that randomly stressing the worldview of software architects with differing approaches to philosophy can help

unlock new approaches to software engineering that are more in line with the hyperliminal environments for which they must design software systems.

6. Conclusions

Giving software engineers the ability to interpret the complex world around them outside of the component metaphor will allow them to better address the risks that software poses to hyperconnected societies.

Residuality theory is one such idea which may help reduce residual causality by helping the designers of software systems to easier identify hyperliminal coupling, by shifting the worldview of the designer from the entrenched component metaphor. It provides an approach that allows, or even forces, software architects to articulate how they are dealing with uncertainty in the complex environments in which they are working.

Residuality theory and other ideas that might emerge from this approach can potentially address the theoretical gap on software and uncertainty described in this paper. This investigation could potentially open up a rich, new seam of ideas and methodologies within the world of software engineering research. This may help software architects to better question and critique passing trends and ideas in the software industry, as well as the ability to more honestly talk in terms of uncertainty and what they do not know as they make decisions. It may also improve the ability of software architects to communicate with each other as a profession around the topic of uncertainty, and the role of the architect in an increasingly uncertain world.

Further areas of research are the confirmation of residuality theory as a viable approach to mitigating unknown sources of stress in software applications, as well as studying the underlying paradigmatic assumptions of software engineers in relation to success and failure in software engineering.

Funding: This research received no additional funding.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: Not applicable.

Acknowledgments: I would like to thank Tanya O'Reilly and Riccardo Bennet-Lovesy for their reviews, editing, and encouragement. Jeffrey Johnson, Jane M Bromley and Anthony Lucas-Smith for their supervision and contribution to the learning journey described here. I would also like to take this chance to mention the huge influence of the ideas of Ralph Stacey who passed away as this article was being formed.

Conflicts of Interest: The author declares no conflict of interest.

References

1. Brubaker, R. Digital hyperconnectivity and the self. *Theor. Soc.* **2020**, *49*, 771–801. [CrossRef]
2. Jesús, J.M. Cyber Risks: Liability and Insurance. The Extraordinary Risks in a Hyperconnectivity World. InDret. 2019, Volume 2. Available online: https://ssrn.com/abstract=3414879 (accessed on 1 April 2019).
3. COOP. Available online: https://www.reuters.com/technology/coop-other-ransomware-hit-firms-could-take-weeks-recover-say-experts-2021-07-05/ (accessed on 20 September 2021).
4. O'Reilly, B.M. The Philosophy of Residuality Theory. *Procedia Comput. Sci.* **2021**, *184*, 809–816. [CrossRef]
5. Xu, Z.; Elomri, A.; Kerbache, L.; El Omri, A. Impacts of COVID-19 on Global Supply Chains: Facts and Perspectives. *IEEE Eng. Manag. Rev.* **2020**, *48*, 153–166. [CrossRef]
6. Huge Challenge Ahead to Get Software Systems Brexit-Ready. Available online: https://www.iod.com/news/navigating-brexit-for-business/articles/huge-challenge-ahead-to-get-software-systems-brexit-ready (accessed on 20 September 2021).
7. Jones, K.H. Engineering antifragile systems: A change in design philosophy. *Procedia Comput. Sci.* **2014**, *32*, 870–875. [CrossRef]
8. Peters, O. The ergodicity problem in economics. *Nat. Phys.* **2019**, *15*, 1216–1221. [CrossRef]
9. O'Reilly, B.M. *Hyperliminal Coupling, Why Software Projects Fail Repeatedly*; Cutter Consortium: Arlington, MA, USA, 2021.
10. Taleb, N.N. *Fooled by Randomness*; Penguin Books: Harlow, England, UK, 2007.
11. Taleb, N.N. *Antifragile: Things That Gain from Disorder*; Random House Incorporated: New York, NY, USA, 2012; Volume 3.
12. Snowden, D.J.; Boone, M.E. A leader's framework for decision making. *Harv. Bus. Rev.* **2007**, *11*, 68.

13. Stacey, R.D. *Complexity and Organizational Reality: Uncertainty and the Need to Rethink Management after the Collapse of Investment Capitalism*; Routledge: Abingdon, UK, 2009.
14. Bar-Yam, Y. When Systems Engineering Fails—Toward Complex Systems Engineering. *Int. Conf. Syst. Man Cybern.* **2003**, *2*, 2021–2028. [CrossRef]
15. Calvano, C.N.; John, P. Systems engineering in an age of complexity. *Syst. Eng.* **2004**, *7*, 25–34. [CrossRef]
16. Leveson, N. Safety III: A Systems Approach to Safety and Resilience. MIT Systems Engineering Lab. Available online: http://sunnyday.mit.edu/safety-3.pdf (accessed on 20 September 2021).
17. Hollnagel, E.; Woods, D.D.; Leveson, N. (Eds.) *Resilience Engineering: Concepts and Precepts*; Ashgate Publishing, Ltd.: Farnham, UK, 2006.
18. Woods, D.D. The theory of graceful extensibility: Basic rules that govern adaptive systems. *Environ. Syst. Decis.* **2018**, *38*, 433–457. [CrossRef]
19. Hole, K.J. *Anti-Fragile ICT Systems*; Springer Nature: Basingstoke, UK, 2016.
20. Kazman, R.; Klein, M.; Clements, P. *ATAM: Method for Architecture Evaluation*; Carnegie-Mellon University, Pittsburgh PA Software Engineering Institute: Pittsburgh, PA, USA, 2020.
21. Principles of Chaos Engineering. Available online: https://principlesofchaos.org (accessed on 20 September 2021).
22. Löwy, J. *Righting Software: A Method for System and Project Design*; Addison-Wesley Professional: Boston, MA, USA, 2019.
23. Ziv, H.; Richardson, D. The Uncertainty Principle in Software Engineering. Available online: http://jeffsutherland.org/papers/zivchaos.html (accessed on 20 September 2021).
24. Klein, G.; Snowden, D. Anticipatory Thinking. Informed by Knowledge Expert Performance in Complex Situations. 2011. Available online: https://www.fs.usda.gov/rmrs/sites/default/files/Klein%20et%20al%20-%20Anticipatory%20Thinking.pdf (accessed on 20 September 2021). [CrossRef]
25. Evans, E.; Evans, E.J. *Domain-Driven Design: Tackling Complexity in the Heart of Software*; Addison-Wesley Professional: Boston, MA, USA, 2004.
26. Robertson, S.; Robertson, J. *Mastering the Requirements Process: Getting Requirements Right*, 3rd ed.; Addison-Wesley Professional: Boston, MA, USA, 2021.
27. Patriarca, R.; Bergström, J.; Di Gravio, G.; Costantino, F. Resilience engineering: Current status of the research and future challenges. *Saf. Sci.* **2018**, *102*, 79–100. [CrossRef]
28. Fowler, M. *Patterns of Enterprise Application Architecture: Pattern Enterpr Applica Arch*; Addison-Wesley: Boston, MA, USA, 2012.
29. Gamma, E.; Helm, R.; Johnson, R.; Vlissides, J.; Patterns, D. *Elements of Reusable Object-Oriented Software*; Design Patterns; Addison-Wesley Publishing Company: Boston, MA, USA, 1995.
30. Martin, R.C. Design principles and design patterns. *Object Mentor* **2000**, *1*, 597.
31. Larman, C. *Object-Oriented Analysis and Design. Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and the Unified Process*; Pearson education India: London, UK, 2002; Volume 10–11, p. 215.
32. Thomas, D.; Andrew, H. *The Pragmatic Programmer: Your Journey to Mastery*; Addison-Wesley Professional: Boston, MA, USA, 2019.
33. Ralph, P.; Tempero, E. Characteristics of decision-making during coding. In Proceedings of the 20th International Conference on Evaluation and Assessment in Software Engineering, Limerick, Ireland, 1–3 June 2016; pp. 1–10.
34. Parnas, D.L. On the criteria to be used in decomposing systems into modules. In *Pioneers and Their Contributions to Software Engineering*; Springer: Berlin/Heidelberg, Germany, 1972; pp. 479–498.
35. O'Reilly, B.M. An introduction to residuality theory: Software design heuristics for complex systems. *Procedia Comput. Sci.* **2020**, *170*, 875–880. [CrossRef]
36. Kauffman, S.A. *The Origins of Order: Self-Organization and Selection in Evolution*; Oxford University Press: Oxford, MA, USA, 1993.
37. Ishimatsu, T.; Leveson, N.G.; Thomas, J.; Katahira, M.; Miyamoto, Y.; Nakao, H. Modeling and Hazard Analysis Using STPA. Available online: http://hdl.handle.net/1721.1/79639 (accessed on 20 September 2021).
38. Stamatis, D.H. *Failure Mode and Effect Analysis: FMEA from Theory to Execution*; Quality Press: Milwaukee, WI, USA, 2003.
39. Heidegger, M. The question concerning technology. *Technol. Values Essent. Read.* **1954**, *99*, 113.
40. Dearmont, D. Transactions of the Charles S. *Peirce Soc.* **1995**, *31*, 185–204.
41. Prigogine, I. The philosophy of instability. *Futures* **1989**, *21*, 396–400. [CrossRef]
42. Latour, B. The Enlightenment without the critique: A word on Michel Serres' philosophy. *R. Inst. Philos. Suppl.* **1987**, *21*, 83–97. [CrossRef]
43. Latour, B. *Science in Action: How to Follow Scientists and Engineers through Society*; Harvard University Press: Cambridge, MA, USA, 1987.
44. Baudrillard, J. *Simulacra and Simulation*; University of Michigan Press: Ann Arbor, MI, USA, 1994.
45. Standish Group. Chaos Report 1995. 2016. Available online: https://www.standishgroup.com/sample_research_files/CHAOSReport2015-Final.pdf (accessed on 20 September 2021).