# The Architecture of Complexity Revisited: Design Primitives for Ultra-Large-Scale Systems

Rick Kazman, Hong-Mei Chen
Department of Information Technology Management
University of Hawaii, Honolulu, HI
{kazman, hmchen}@hawaii.edu

## Abstract

*As software-intensive systems continue to grow in scale and complexity the techniques that we have used to design and analyze them in the past no longer suffice. In this paper we look at examples of existing ultra-large-scale systems—systems of enormous size and complexity. We examine instances of such systems that have arisen spontaneously in nature and those that have been human-constructed. We distill from these example systems the design primitives that underlie them. We capture these design primitives as a set of tactics— fundamental architectural building-blocks—and argue that to efficiently build and analyze such systems in the future we should strongly consider employing such building-blocks.*

**Keywords:** software architecture; architecture design; ultra-large-scale systems; design tactics; wicked systems.

## 1. Introduction

In 1962 Herbert Simon wrote about properties of "complex systems". He believed that such properties would be relevant to "complex systems that are observed in the social, biological, and physical sciences" [28]. The central theme of his work was that "complexity frequently takes the form of hierarchy" with stable sub-assemblies and this structure is critical for managing complexity. To control complexity, these sub-systems are "nearly-decomposable" which means that there is more interaction within a sub-system than between sub-systems, and there are few types of sub-systems. Biological systems are inherently hierarchical, as Simon argued: cells are organized into tissues, tissues into organs, organs into individuals. Social systems are also hierarchical: people are grouped into families which are part of tribes, which are part of cultures, and so forth.

In the five decades since Simon's paper, we have seen "ultra-large-scale" (ULS) systems [24] arise in society: power grids, telephone systems, and the internet, for example. While such systems are undoubtedly hierarchical and nearly-decomposable, they possess other very important properties, and these properties do not arise from hierarchy and near-decomposability alone.

Ultra-large-scale systems are: "Ultra-large-scale systems are interdependent webs of software, people, policies, and economics" [16]. ULS systems are characterized by extreme size along multiple dimensions: data volumes, amount of hardware and software components, number of lines of code, and numbers of users. There are seven important characteristics that have been determined to distinguish ULS systems from the majority of software-intensive systems today [24]: 1) decentralization, 2) inherently conflicting, unknowable, and diverse requirements, 3) continuous evolution and deployment, 4) heterogeneous, inconsistent, and changing elements, 5) erosion of the people/system boundary, 6) normal failures, and 7) new paradigms for acquisition and policy. The ULS report explains and justifies each of these characteristics and goes on to explain that: "These characteristics are beginning to emerge in today's DoD systems of systems; in ULS systems they will dominate" [24]. The dominance of these characteristics is why we must reason about and

URI: https://hdl.handle.net/10125/103477
978-0-9981331-6-4

design ULS systems differently than the vast majority of existing systems.

Consider, for example, the ramifications of the characteristic of "normal failures". If a CPU fails, on average, once every three years and you have 100,000 CPUs in your server farm (which is not particularly large) then, on average, 300 of these will fail on any given day. Failure is not a crisis; failure becomes a normal, predictable occurrence that you can and should plan for. As emphasized by the ULS report, "scale changes everything" [24].

Such characteristics mean that ULS systems of the future cannot be designed in ways that have hitherto sufficed for systems of precedented scale. While it is true that some ULS systems exist today—e.g. the internet and our telephone system—such systems have evolved over decades. As architects, we would like to be able to design future ULS systems efficiently and effectively, rather than by trial-and-error over decades. And we are beginning to see systems emerge that possess ULS characteristics: social networks, peer-to-peer (P2P) systems, vehicle systems, the Smart Grid, and large networks of IoT and edge devices. The successful instances of these kinds of systems of the future will need to be designed in ways that are fundamentally different from the way that the majority of software systems are designed today.

If ULS systems are our future, and if they are qualitatively and quantitatively different from almost all existing systems, what are the fundamental primitives that we can employ to design, and reason about and analyze, their architectures? Given that most existing systems lack the characteristics of ULS systems and yet these systems already do not scale well and have rampant technical debt [12], it is evident that existing design techniques and practices are not entirely suitable for ULS systems. ULS systems must be designed differently. In what follows we describe the ways that ULS architectures differ from "traditional" system architectures, and we will present a set of primitives for thinking about and designing such systems.

## 2. Background

ULS systems are generally viewed as being "wicked" problems [26]. Wicked problems have been studied for decades. They possess several unique characteristics that make them wicked; that is difficult to design, manage, and evolved:

- stakeholders do not all agree on the problem to be solved—requirements are vague and unstable
- solutions are not right or wrong, they are better or worse
- enormous complexity, both among the subcomponents and between the "problem" and the world; and any solution may change the problem
- they have no single objective measure of success

For example the internet is arguably the largest man-made system ever created, and its design can been viewed as a wicked problem: there is no single objective measure of success, the internet and its requirements are always changing, and its stakeholders are constantly pulling it in diverse directions.

Being wicked problems, ULS systems undermine our existing assumptions surrounding software, particularly how to create and maintain it. For example, it is not possible to resolve all requirements conflicts. And ULS systems cannot be developed using lifecycle models and design primitives that have sufficed in the past [18].

## 3. Architectural Tactics

Architectural tactics are design primitives. A tactic is a design decision that influences the achievement of a desired quality attribute response; that is, tactics directly affect the system's response to some stimulus. When an architect makes design decisions they are, knowingly or not, making decisions about tactics. For example, consider the tactics hierarchy for availability, shown in Figure 1 ([3], [27]).

An architect who wants to design a system to meet high availability requirements needs to consider and make decisions about how to detect faults, how to recover from faults, how to reintroduce recovered or replacement components, and how to prevent faults from occurring in the first place. These are the categories of architectural design decisions that need to be made. Within each category are the specific tactics that an architect can choose. These tactics are, in turn, realized by patterns or tools or frameworks, or directly in code.

We do not "invent" tactics. We merely attempt to distill and catalog the design techniques that have proven themselves to be most effective, across systems and over time.

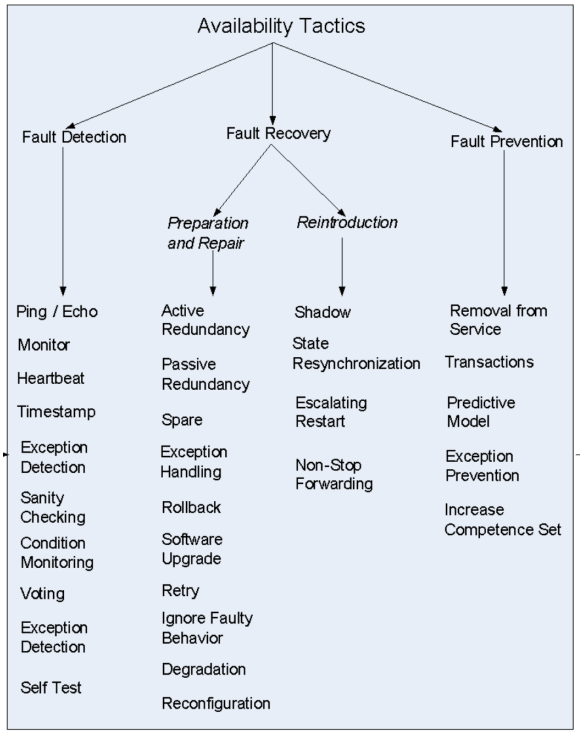


Figure 2: Availability Tactics

To discover the architectural primitives—the tactics—that underlie ULS systems we needed to examine the broadest possible range of systems that exhibit ULS properties—P2P systems (such as Skype and its predecessor KaZaA and, more recently, blockchain-based systems), the internet, and the public switched telephone network (PSTN), as well as a number of biological systems—and attempt to determine whether they have anything in common. If so then our goal is to distill their common design approaches. This research approach may seem somewhat unorthodox because the final example—biological systems—is neither human nor it is computational. But such systems—ant colonies, termite colonies, slime molds—are ultra-large-scale systems and exhibit many desirable properties such as self-organization, emergent behavior, and adaptability, despite being without centralized control, continuously changing, and suffering normal failures. And natural systems, such as the brain and ant colonies, have been the inspiration for the designs of many artificial systems in the past.

All ultra-complex systems—human societies, ant colonies, the internet, the power grid, slime molds—consist of a set of peers: people, ants, power stations, nodes, servers, etc. Each peer controls, to some extent, its own resources and its destiny. These peers contain subsystems that may be composed into more complex structures. So each peer is itself a hierarchical structure [28].

From these example systems we infer that architectures for ULS embody three basic design approaches:

- *peer to peer structure*: to achieve ultra-large scale, dependence upon centralized resources must be avoided. This restriction implies some form of P2P structure, where peers provide their own resources and exert control over their destiny. Peers may organize themselves into more complex units, leading to the next design approach. Communities of humans and ecosystems are examples of P2P systems in nature. Systems of systems, agent-based systems [13] and the internet are examples in computational domains.
- *local assemblies of components*: complex systems contain local assemblies of a "small" number of components. These local assemblies may be temporary or permanent but there are typically many of them in existence at any given time. Assemblies are independent from, or interact weakly with, other assemblies.
- *hierarchical structure*: Simon argued for hierarchical structure as an organizing principle of complex systems. Systems of systems (which contain systems which contain sub-systems), internet domains, and layered systems (which contain modules) are familiar examples of hierarchical structure in the computational domain.

These design approaches have a number of important ramifications for the creation of ULS architectures, as we will show.

## 3.1. The Architecture of KaZaA

KaZaA, and its successor, Skype, are both distributed systems that largely rely on resources "donated" by participants. They achieved enormous scalability and high performance despite having few

centralized, controllable resources. In a study of KaZaA [23] the authors describe a set of architectural principles:

1. *Distributed Design*: all computation runs on servers provided by the peers, so no infrastructure servers. This ensures no centralization, resilience to faults and attacks, built-in scalability.
2. *Exploiting Heterogeneity*: peers have different availability, bandwidth, CPU power, etc. This makes it possible to have different classes of servers providing different services.
3. *Load Balancing*: by keeping the "degree" of each node approximately the same (i.e. coordination with the same number of neighbors) load can be approximately balanced.
4. *Locality in Neighbor Selection*: a neighbor is close in terms of network topology and latency; a node should be able to coordinate with neighbors quickly.
5. *Connection shuffling*: although the size of the neighbor group remains fairly stable over time and across the network, the actual content of the neighbor group for any node is dynamic.
6. *Efficient gossiping algorithms*: to promote connection shuffling and find new peers as they enter the network, nodes must "gossip" with each other frequently, learning about their environment.

The design of these systems has been highly successful. KaZaA in its heyday had over 3 million active users. Skype reportedly hosted 40 million or more users a day, on average, in 2020, according to Microsoft.[1]

## 3.2. Biological Systems

Biological systems—termite colonies, ant colonies, slime molds—are arguably ULS systems and all exhibit desirable characteristics ([4], [5], [6], [10], [11], [14]): without any central control, they manage to react in purposeful ways. They learn, adapt and evolve, and operate in a robust manner in the face of uncertain and incomplete information. The success of biological systems should be studied, to understand their organizational principles and success factors.

Flake [14] outlines three attributes that distinguish agents in complex systems:

1. Collections, Multiplicity, and Parallelism: complex systems are made up of large collections of agents working in parallel. These agents contain small amounts of variation that allow them to seek different solutions to the same problem, or react to different environmental stimuli.
2. Iteration, Recursion, and Feedback: agents iterate, perhaps by reproducing, so that they persist over time. Complex systems contain substantial amounts of self-similar structure owing to recursion. And agents not only react to changes in their environment, but often change their environment.
3. Adaptation, Learning, and Evolution: the inevitable finiteness of resources, in combination with parallelism and iteration, cause agents to need to adapt. Successful adaptations further iterate while unsuccessful ones wither.

Holland, in his book on complex adaptive systems (CASs) [17] analyzes how complexity arises from adaptation. He defined a set of seven properties that characterize CASs. These are:

1. Aggregation: collections of agents have their own properties and behaviors
2. Tagging: tags support aggregation and boundary formation
3. Non-linearity: the behavior of aggregates is more complex than the sum of its parts.
4. Flows: information and, in the real world, goods, flow between agents
5. Diversity: agents are diverse, filling different roles in their environment and often adapting to fill new roles
6. Internal models: agents contain models of their environment and constantly update those models to reflect observations
7. Building blocks: an agent's internal model can be decomposed by a small set of building blocks that can be flexibly combined

Holland divides these properties into characteristics (Aggregation; Nonlinearity; Flows; Diversity) and mechanisms (Tagging; Internal

[1] https://www.microsoft.com/en-us/microsoft-365/blog/2020/03/30/introducing-new-microsoft-365-personal-family-subscriptions/

Models; Building blocks). The mechanisms have particular importance as classes of architectural primitives, as we will see.

Ant Colony Optimization (ACO) [11] is a computational technique for searching—finding paths through graphs—inspired by the behavior of ant colonies. Ants do this by laying down pheromone trails as they explore, where other ants follow those paths. Paths that are useful—e.g. leading to food—are reinforced by many ants and become highly traveled. Ones that are not useful are not reinforced; as the pheromones evaporate such paths are abandoned. ACO simulates this behavior, by not only following existing paths but also randomly trying new paths and leaving computational "pheromones" that decay over time. Using this technique, ACO has been found near-optimal solutions to the traveling salesman problem, for example. And ACO is robust in the face of unexpected changes. For example, in a routing problem if new destinations are introduced or old ones are removed, the algorithm will quickly adapt, just as an ant colony reacts to a new source of food or source of danger.

### 3.3. The Architecture of the Internet

The internet is arguably the largest, most successful man-made ULS system ever. In 1996 the IETF [25] articulated the design foundations of the internet, as they had evolved. The architectural principles they distilled are:

1. *One and only one protocol*: this supports uniform and relatively seamless operations in a competitive, multi-vendor public network.
2. *End-to-end functions realized by end-to-end protocols*: "The network's job is to transmit datagrams as efficiently and flexibly as possible. Everything else should be done at the fringes." This means that the system maintains little state; just enough to keep the core smoothly working, e.g. routing info, QoS guarantees, etc.
3. *Heterogeneity:* is inevitable and must be supported by design. (Peers will be constructed differently, but must nonetheless interoperate.)
4. *Scale-free design*: all design decisions must support scaling to many nodes per site and many millions of nodes.
5. *Modularity:* keep design decisions and implementations modular whenever possible (for example internet nodes are largely autonomous).
6. *Send/receive asymmetry*: Be strict when sending and tolerant when receiving.
7. *Self-description*: Objects should be self-describing.

Many of these properties *evolved* over the years to become the stable and highly effective infrastructure we know today.

### 3.4. The Public Switched Telephone Network

The Public Switched Telephone Network (PSTN) is one of the longest-lived, largest, most scalable, and most robust systems ever created ([15], [29]). The PSTN successfully handles hundreds of millions of concurrent customers with switches that experience no more than 2 hours of failure over 40 year lifespans. An analysis of PSTN failures [21] characterized a number of properties that led to its robustness, despite its size and complexity:

1. Reliable software: the underlying software seldom fails, and up to half of the code is devoted to error detection and correction

2. Dynamic rerouting: because network failures are typically localized, the system dynamically routes around failures. Small outages in the system's functioning are typically not catastrophic.

3. Loose coupling: components are loosely coupled, knowing little about each other and each other's internal state

4. Human intervention: when all else fails, the PSTN system has operators who can intervene.

In the PSTN these principles have also evolved over decades, via the engineering efforts and trial-and-error of many organizations.

### 3.5. Distilling ULS Architectural Tactics

From these examples we now attempt to examine their commonalities, to distill a set of generic architectural mechanisms that can be used when designing systems of ultra-large-scale. These are primitives that an architect can use when designing a system that is anticipated to grow without bounds, and whose architectures may evolve in unanticipated ways.

As mentioned above, a tactic is a design decision that is influential in controlling a quality attribute response, such as latency, or mean time to failure, or ease of modification. Tactics have previously been described for the quality attributes of availability, deployability, energy efficiency, integrability, modifiability, performance, safety, security, testability, and usability [3]. In prior work we have employed these tactics as an intellectual foundation for designing and analyzing software architectures, and they have been adopted in software engineering curricula and in industrial software design ([7], [22]).

But tactics are not just design primitives for computer-based systems: they apply to complex systems of any kind. The forces and principles of architecture do not change based on the scale or materials of the architecture being created. In fact, the variety of systems considered above, and the small number of organizational principles and design primitives found, is striking evidence for the existence of ULS tactics, as we now describe.

## 4. Tactics for ULS Systems

A set of tactics for designing scalable (ULS) systems is presented in Figure 2. These tactics are divided into three categories: *Building Blocks, Aggregation,* and *Interaction*. This categorization roughly follows Holland's "mechanisms" for CASs [17]: building blocks, tagging (which allows for interaction), and internal models (which support aggregation). *Building blocks* are the foundational structures for creating the peers that comprise a ULS system. *Aggregation* allows a designer to create collections of peers, at arbitrary scale, and *Interaction* allows the assemblies to interact with each other and their environment. Any architect, when making design decisions for a ULS system, needs to consider all three of these categories.

### 4.1. Building Blocks

Building Blocks are the techniques for creating the lowest-level system elements, the raw materials from which all ULS systems are constructed. There are three tactics in this category: Modularity, Self-description, and Environment Models.

Modularity: is arguably the most powerful design principle in software engineering and its importance in a ULS system cannot be overemphasized. Modularity has been argued to support super-linear growth in software, for example [20]. Modularity is a core property in the architecture of the internet [25] and it is what supports the loose coupling of the PSTN [21]. Biological systems ([14], [17]) are composed of autonomous agents: each is a module of sorts, interacting via some "public interface", with its internal state unavailable to the external world. Systems of systems contain systems—independent black boxes—each operating and evolving independently [13].

Self-description: is a core property of the internet: "objects should be self-describing" [25]. This allows them to operate relatively autonomously but still interact with their environments. Self-description supports adaptation, learning, and evolution [17] all of which are an agents' updates to their internal models. These models allow independent agents to interact with their environment, and to modify this interaction over time as conditions change.

Environment models: similar to self-description—which are models that agents maintain of themselves—are environment models, which are models that agents maintain of their environment. CASs explicitly model their environment and these models are constantly updated to reflect observed phenomena . For example, P2P systems [23] use such models in their gossiping algorithms to maintain a description of their environment.

### 4.2. Aggregation

There are four tactics within Aggregation: Self-similar structure, Heterogeneity, Concurrency, and Abstract Connections.

Self-similar structure: to grow without bound, systems must treat collections of entities similar to individual agents. This property—sometimes called "scale invariance"—makes it easy for the system to be self-configuring and self-adapting, dynamically responding to changes in availability, opportunity, and processing power. This is a realization of the "collections" property of biological systems [14] and is crucial in the architecture P2P networks, where super nodes are often simply "promoted" versions of ordinary nodes [18]. In fact, in the internet there is no distinction between a connection to an important node—a hub—and a small, unimportant "dead-end" node; it has a fractal structure.

Fisher [13] makes it clear that emergent systems must treat each other homogenously, irrespective of their scale: "For each node, a node-centric perspective captures the current structure of

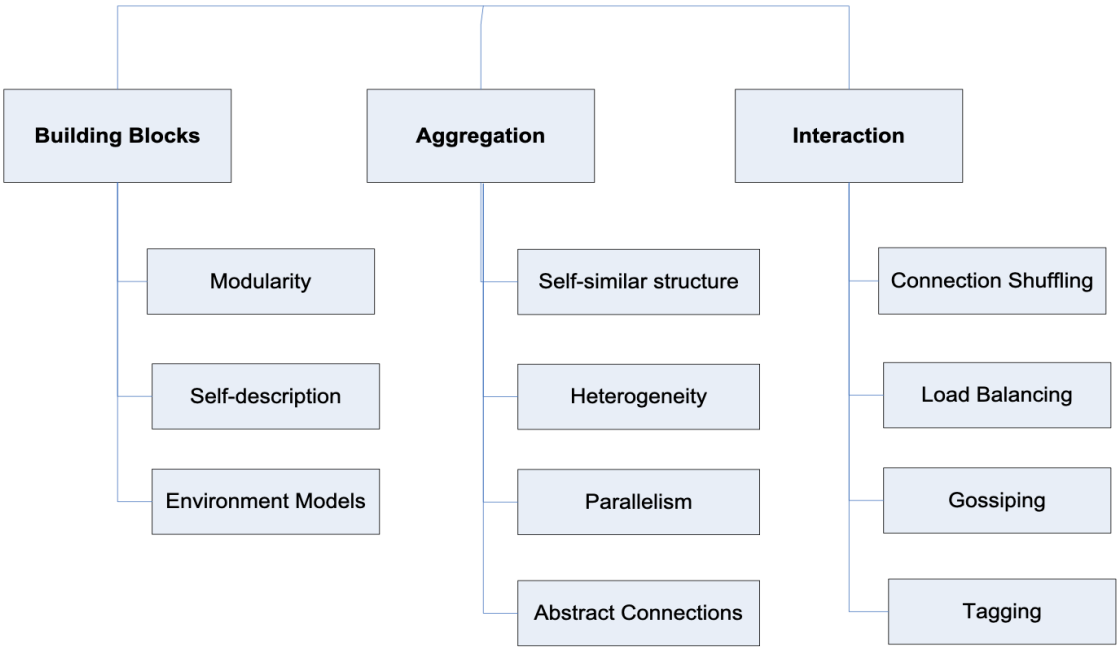


Figure 2. Tactics for ULS Systems

interconnection and interaction with its immediate neighbors and as much information about its neighbors as is useful and obtainable".

Heterogeneity: in a ULS system the peers will have different properties (availability, bandwidth, CPU, etc.) which must be supported by the design. This is a principle in the internet [25] in the design of KaZaA [23] in CASs (where it is termed "diversity") [17] and in systems of systems [13]. The design must support heterogeneity by abstracting characteristics of peers that are important for interaction. In this way the system and its components become insensitive to heterogeneity.

Concurrency: means that all peers run independently, at the same time, without any centralized control. This property, along with heterogeneity, ensures that there is no single point or mode of failure, and it allows a system to scale without bounds as each peer, running in parallel, brings its own resources. This is clearly a core property of biological systems [14], the internet [25], systems of systems [13], and P2P systems [23].

Abstract Connections: for aggregates to be able to grow without bounds, connections must be abstract, and realized at run-time. This means that nodes must avoid hard-wiring anything: neighbors, buffer sizes, connection pool sizes, etc. This is related to the notion of "scale-free design" [25], and this is why systems of systems treat each other simply as "nodes" annotated with properties [13]. In biological systems, connections are all realized at run-time—nothing is hard-coded. Slime molds, for example, connect via the release of cyclic adenosine monophosphate—an intra-cell messenger [5].

### 4.3. Interaction

There are four tactics within Interaction: Connection shuffling, Load balancing, Gossiping, and Tagging.

Connection shuffling: for interaction to remain robust over time, it is important to not only avoid hard-wiring connections, but to continuously seek out new connections. In this way the system remains robust with respect to failures of individual peers or connections, and can optimize its behavior with respect to the availability and topology of resources. This is a core feature of the PSTN (called "dynamic rerouting" [21]), of P2P systems [23], and of the internet [25]. Ant colonies employ randomness in the

actions of individual ants, so that a small portion of them get "lost" regularly, and discover new (potentially better) routes to food [10].

Load balancing: for a ULS system to work efficiently, work needs to be appropriately balanced among peers. This is a basic design approach found in P2P networks [23], and has been much researched by the P2P community. In some ULS systems this allocation follows a power law [2], where most nodes have only a few neighbors and hence have only a small amount of work, but a few nodes have enormous numbers of neighbors. For example, the major internet exchange points (IXPs) have many hundreds or even thousands of neighbors.

Gossiping: nodes need to interact to adapt to their ever-changing states and environments. Neighboring nodes constantly "gossip", exchanging topological and task-specific information. In the internet this information is primarily routing tables. In P2P systems this includes information about the data or services a node provides. Biological systems achieve information exchange—albeit indirectly—via "stigmergy": an ant's pheromone trails allow coordination amongst the independent agents in the colony [5]. Gossiping allows a node to update its internal or external model and from this to adapt to its environment.

Tagging: to allow groups to form, and to create boundaries among groups, tagging is required. Tagging supports the creation of "collections" as meaningful named entities [17]. In the internet, DNS servers are named (tagged) entities that support a hierarchical naming system (where name servers pass requests for sub-domains to sub-servers). Super-nodes in P2P systems are ordinary nodes that are promoted, and then they are tagged as such. In ant colonies, nest-mates recognize each other via a set of biological tags [6].

### 4.4. Consequences

Not only are each of these tactics characteristic of complex systems but they are necessary characteristics: it is unimaginable that a system could grow without bound without all of these tactics. Different tactics may be present to different degrees in the PSTN, or the internet, or biological systems, but they are all present.

Such a claim cannot be proven, but we can cite evidence from systems that we did not initially consider to justify that these tactics are indeed foundational. Next we discuss two examples—slime molds (natural) and Mobile Ad Hoc Networks (artificial)—to argue the claim that these tactics are *necessary* building blocks for ULS systems.

Consider the remarkable adaptability of slime molds [4]. Each slime mold cell is a peer and hence modular, with its own state, hiding its internals. These cells operate in parallel. They clearly exist as individuals, but slime mold cells can also aggregate into groups of up to 105 organisms to create "slugs" that can travel, or into spore-bearing fruiting structures—called sporangiophore—for reproduction. These cells have internal models, as they are capable of fully independent lives—feeding and reproducing in parallel. Slime mold cells have environmental models—for example, they shy away from light, and they change state under stressful conditions (such as a lack of food, or a change in pH). Also slime mold cells exhibit heterogeneity: some cells form the stalk of the sporangiophore and others specialize to become spores; and these cells become cysts when conditions turn unfavorable.

In the artificial domain, consider MANETs (Mobile Ad hoc NETworks). MANETs are defined as "a collection of mobile nodes, communicating among themselves over wireless links and thereby forming a dynamic, arbitrary graph" [8]. Each node is a distinct wireless device, and therefore modular. Because each node is independent, they can operate in parallel. Such nodes are typically heterogeneous, sharing perhaps no more than a common (dynamic) communication protocol, which acts as an abstract connection mechanism. Nodes do not expose internals; they interact via well-defined interfaces. To participate in a network, a MANET node must be self-describing, representing itself to neighbors, including the set of other nodes reachable through it. MANET nodes exhibit self-similar structure: "a mobile router may attach to any router, including another mobile router, forming networks of mobile routers to an arbitrary depth … Commonly the terms 'nested mobile network' or 'nested NEMO' are used for this situation". Because these networks are mobile, nodes are constantly moving into and out of range, and so nodes are continually connection shuffling and gossiping.

In short, MANETs and slime molds have achieved their remarkable properties of scale because they instantiate, in their own unique ways, the set of scalability tactics presented in Figure 2.

## 5. Conclusions

ULS systems are becoming more common as the world becomes more digitized and more interconnected. This trend shows no sign of slowing. Since the original ULS report was published in 2006, ultra-large systems have been built and deployed in increasing numbers. But they remain a challenge for architects because few software and systems design professionals have a deep well of experience building systems of ultra-large scale. The purpose of this paper, as with the purpose of all research into architectural tactics and patterns, is to codify the knowledge contained in the heads of the very best architects so that others with less skills or experience can more confidently create architectures with predictable properties, predictable risk and cost.

Architectural tactics are a kind of "periodic table" of design—elemental design operations. Tactics are not novel research artifacts; tactics, like design patterns, arise repeatedly in practice. Every architectural tactic exists to serve an objective in the process of design, to enable or to control a systemic response. The tactics presented here primarily exist to enable and manage scalability concerns for a broad range of ULS systems and each of them comes with a pedigree, having been used over and over again in successful natural and artificial systems of ultra-large scale.

Tactics are not, however, a panacea. An architect still needs to understand them, and reason about their pros, cons, and tradeoffs. For example, the tactic of Heterogeneity increases the potential size of the pool of available resources, which is good. It also limits the effects of common-mode failures, which is good. But this comes at a cost of having to deal with this heterogeneity, which might increase maintenance or deployment costs. Environment models allow each node to respond independently to changes, avoiding the need to centralized monitoring and control, which is good. But such models, if not designed with scalability in mind, can be a bottleneck or single source of failure. Connection shuffling ensures that a ULS system does not get "stuck" in rigid forms of behavior, which will, over time, inevitably become sub-optimal; the system continuously adapts by inserting randomness into its behavior. But this benefit comes at the cost of a small reduction in run-time performance.

Taken together, these ULS tactics form an ontology of design for ULS systems of the future. The ontology was created by triangulating from examples in a diverse set of domains—natural and artificial—that exhibit ultra-large-scale. While it is likely that these tactics are incomplete, as any set of design primitives will evolve over time—they do concisely capture a wide range of scalability phenomena relevant to the "wicked" problems that we face in developing complex systems today, and that we will increasingly face in the future.

Why should you care? There are two obvious practical uses for this set of tactics (and, in fact, for all tactics categorizations):

1. If you are the architect of a system that may experience explosive growth then you can use a set of tactics as a guide for making design choices.

2. If you are analyzing such a system, you can use these tactics as a questionnaire, to focus and guide your analysis efforts.

These tactics, we claim, can help to make the incredibly risky process of design in a ULS system—which is, by its very nature, a wicked system—a bit more systematic and a bit less risky.

## 6. References


[1] Ahmadjee, S., Mera-Gomez, C. Bahsoon, R. Kazman, R. (2022) "A Study on Blockchain Architecture Design Decisions and their Security Attacks and Threats", *ACM Transactions on Software Engineering and Methodology*, 31:2, April, 2022.

[2] Barabasi, A-L. (2003) *Linked: How Everything Is Connected to Everything Else and What It Means for Business, Science, and Everyday Life*, Basic Books.

[3] Bass, L., Clements, P., Kazman, R. (2021) *Software Architecture in Practice*, 4th ed., Addison-Wesley.

[4] Bryden, J. (2005) "Slime Mould and the Transition to Multicellularity: The Role of the Macrocyst Stage", *Advances in Artificial Life*, Springer, 551-561.

[5] Camazine, S., Deneubourg, J-L, Franks, N. R., Sneyd, J., Theraula, G., Bonabeau, E. (2001) *Self-Organization in Biological Systems*, Princeton University Press.

[6] Carlin, N., and Hölldobler, B. (1986) "The Kin Recognition System of Carpenter Ants", *Behavioral Ecology and Sociobiology*, (19:2), 123-134.

[7] Chen, H., Kazman, R, Haziyev, S. (2016) "Agile Big Data Analytics for Web-based Systems: An Architecture-centric Approach", *IEEE Transactions on Big Data*, 2:3, Sept. 2016, 234-248.
[8] Clausen, T. *A MANET Architecture Model*, INRIA Research Report No. 6145, January 2007.
[9] Cervantes, H. and Kazman, R. (2016) *Designing Software Architectures: A Practical Approach*, Addison-Wesley.
[10] Detrain, C., and Deneubourg, J-L. (2006) "Self-organized structures in a superorganism: Do ants "behave" like molecules?", *Physics of Life Reviews* 3, 162–187.
[11] Dorigo, M., and Stützle, T. (2004) *Ant Colony Optimization*, MIT Press.
[12] Ernst, N, Delange, J, and Kazman, R. (2021) *Technical Debt in Practice—How to Find It and Fix It*, MIT Press.
[13] Fisher, D. (2006) "An Emergent Perspective on Interoperation in Systems of Systems", Software Engineering Institute Technical Report CMU/SEI-2006-TR-003.
[14] Flake, G. (1998) *The Computational Beauty of Nature*, MIT Press.
[15] Hanmer, R. (2007) *Patterns of Fault-Tolerant Software*, Wiley.
[16] Hissam, S., Klein, M., Moreno, G., Northrop, L., Wrage, L. (2016), "Ultra-Large-Scale Systems: Socio-adaptive Systems", Software Engineering Institute White Paper, https://resources.sei.cmu.edu/library/asset-view.cfm?assetid=493859.
[17] Holland, J. (1996) *Hidden Order: How Adaptation Builds Complexity*, Basic Books.
[18] Kazman, R., and Chen, H-M. (2009) "The Metropolis Model: A New Logic for the Development of Crowdsourced Systems", *Communications of the ACM*, July 2009.
[19] Kazman, R., Haziyev, S., Yakuba, A., Tamburri, D. (2018) "Managing Energy Consumption as an Architectural Quality Attribute", *IEEE Software*, 35:5, September/October 2018.
[20] Koch, S. (2007) "Software evolution in open source projects - A large-scale investigation", *Journal of Software Maintenance and Evolution*, (19:6), 361-382.
[21] Kuhn, D. R. (1997) "Sources of Failure in the Public Switched Telephone Network", *IEEE Computer*, (30:4), April, 1997.
[22] Lattanze, A. (2008) *Architecting Software Intensive Systems: A Practitioners Guide*, CRC Press.
[23] Liang, J., Kumar, R., and Ross, K. (2005) "The KaZaA Overlay: A Measurement Study", *Computer Networks*.
[24] Northrop, L., Feiler, P., Gabriel, R., Goodenough, J., Linger, R., Longstaff, T., Kazman, R., Klein, M., Schmidt, D., Sullivan, K., Wallnau, K. (2006) *Ultra-Large-Scale Systems: The Software Challenge of the Future*. SEI/CMU,.
[25] RFC 1958, (1996) Architectural Principles of the Internet, Internet Engineering Task Force (IETF).
[26] Rittel, H., and Webber, M. (1973) "Dilemmas in a General Theory of Planning", *Policy Sciences*, (4), 155-169.
[27] Scott, J., and Kazman, R. (2009) *Realizing and Refining Architectural Tactics: Availability*, Software Engineering Institute Technical Report CMU/SEI-2009-TR-006.
[28] Simon, H. (1962) "The Architecture of Complexity", *Proceedings of the American Philosophical Society*, (106:6), 467-482.
[29] Utas, G. (2005) *Robust Communications Software: Extreme Availability, Reliability and Scalability for Carrier-Grade Systems*, Wiley.