RE^{X}: A Development Platform and Online Learning Approach for Runtime Emergent Software Systems

Barry Porter[†], Matthew Grieves[†], Roberto Rodrigues Filho[†] and David Leslie[‡]
[†]*School of Computing and Communications;* [‡]*Department of Mathematics and Statistics*
Lancaster University, UK

Abstract: Conventional approaches to self-adaptive software architectures require human experts to specify models, policies and processes by which software can adapt to its environment. We present RE^{X}, a complete platform and online learning approach for *runtime emergent software systems*, in which all decisions about the *assembly* and *adaptation* of software are machine-derived. RE^{X} is built with three major, integrated layers: (i) a novel component-based programming language called Dana, enabling discovered assembly of systems and very low cost adaptation of those systems for dynamic re-assembly; (ii) a perception, assembly and learning framework (PAL) built on Dana, which abstracts emergent software into configurations and perception streams; and (iii) an online learning implementation based on a linear bandit model, which helps solve the search space explosion problem inherent in runtime emergent software. Using an emergent web server as a case study, we show how software can be autonomously self-assembled from discovered parts, and continually optimized over time (by using alternative parts) as it is subjected to different deployment conditions. Our system begins with no knowledge that it is specifically assembling a web server, nor with knowledge of the deployment conditions that may occur at runtime.

1 Introduction

Modern software systems are increasingly complex, and are deployed into increasingly dynamic environments. The result is systems comprising millions of lines of code that are designed, analyzed and maintained by large teams of software developers at significant cost. It is broadly acknowledged that this level of complexity is unsustainable using current practice [15]. In recent years this has driven research in autonomic, self-adaptive and self-organizing software systems [26, 31, 14], aiming to move selected responsibility for system management into the software itself. While showing promise, work to date retains a very high degree of human involvement – either in creating models to describe systems and their adaptation modes [10, 13], policies to control adaptation at runtime [20], or designing and running courses of offline training with available historical data [12]. These are *human-led* approaches to the above complexity problem, designed to fit well with current software development practice.

We push these concepts to their limits with a novel *machine-led* approach, in which a software system autonomously *emerges* from a pool of available building blocks that are provided to it. We demonstrate the first such example of a software system able to rapidly self-assemble into an optimal form, at runtime, using online learning. This is done with no models or architecture specifications, and no policies for adaptation. Instead, the live system learns by assembling itself from needed behaviors and continually perceiving their effectiveness, such as response time or compression ratio, in the environments to which the system is subjected. The building blocks of our approach are based on *micro-variation*: different implementations of small software components such as memory caches with different cache replacement strategies or stream handlers that do or do not use caching. As we use relatively small components, this kind of implementation variant is easy to create. By autonomously assembling systems from these micro-variations, and their various combinations, we then see emergent designs to suit the conditions observed at runtime.

Implemented as a development platform called RE^{X}, we present three major integrated contributions, each a key part of the solution to *emergent computer software*:

- **An implementation platform**: We present the key features of *Dana*, a programming language with which to create small software components that can be assembled into emergent software systems. Dana offers a uniform way to express systems in these terms, and near-zero-cost runtime adaptation.
- **A perception, assembly and learning framework**: We present the details of *PAL*, a framework built with Dana that controls the dynamic discovery and assembly of emergent software, perceives the effectiveness and deployment conditions of that software (such as input patterns and system load characteristics), and feeds perception data to an online learning module.
- **An online learning approach**: We present an application of statistical linear bandits, using Thompson sampling, as an effective online learning algorithm that helps to solve the search space explosion inherent in emergent software. This is done by sharing beliefs about individual components across the configurations in which they can appear.

Using a prototype emergent web server as an example, we show how a system can be autonomously assembled from discovered parts, and how that system can subsequently be optimized to its task by seamlessly reassembling it from alternative parts. We evaluate our approach by subjecting our web server to various usage patterns, demonstrating how different designs rapidly emerge over time as conditions change. This emergence occurs through online learning, based on perception streams that indicate the internal well-being of the software and the external conditions to which it is currently being subjected. We also show how a simple classifier adds "memory" to the system, avoiding re-learning of environmental change. In our current implementation this classifier is manually defined; further automation here is a topic of future work.

Our work paves the way to: (i) significantly reducing human involvement in software development, thereby reducing the scale of modern development processes; and (ii) creating systems that are far more responsive to the actual conditions that they encounter at runtime, therefore offering higher performance in those conditions.

The remainder of this paper is structured as follows. In Sec. 2 we discuss our approach in detail, presenting the above three contributions and how they integrate into a complete platform for emergent computer software. In Sec. 3 we then evaluate the system in terms of its ability to continuously (and rapidly) assemble optimal software compositions as external stimulus changes. In Sec. 4 we discuss related work and we conclude the paper in Sec. 5.

2 Approach

Our approach has three major contributions that build on each other to provide an integrated solution for emergent software. An overview is shown in Fig. 1.

At the bottom layer is our implementation platform (Dana) for creating software components that can be assembled / re-assembled in various ways into different systems. This layer provides an API to control the loading, unloading and interconnection of these components. The upper two layers then contain our perception, assembly and learning (PAL) framework. Specifically, at the middle layer are our assembly and perception modules, responsible for assembling entire systems from available components and perceiving the state of those systems. Together these two modules offer an API to control the way in which the system is currently assembled, and to view the perception streams that the system is emitting. The top layer contains our learning module, which uses the assembly and perception API. This module learns correlations between particular assemblies of behavior and the way that the system perceives its own well-being, under different external stimuli of input patterns or deployment environment conditions such as CPU load.

Our overall approach uses dynamic *micro-variation* of behavior: the ability to continually discover and configure components in and out of a live system that perform the same overall task but do so in different ways. By experimenting with these variations, and their combinations, we see the ideal system emerging over time for the current usage pattern and deployment environment conditions[1].

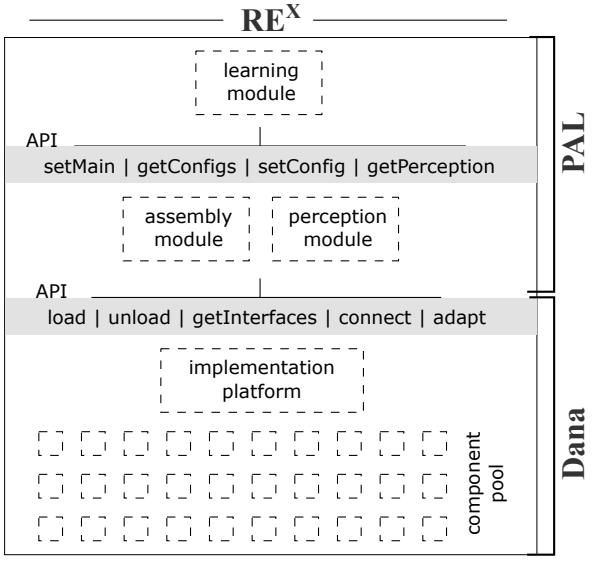

Figure 1 – Overview of our approach.

In the following sections we present Dana; our PAL framework; and the online learning approach used in PAL.

2.1 Dana: An implementation platform for runtime adaptive micro-variation

A platform for emergent software must enable us to build small units of behavior, and to express the interrelationships between them, such that we can create micro-variations of these units that can be autonomously assembled, and seamlessly re-assembled, in a live system. To do this we started from the component-based software development paradigm [29], well-established in forging adaptive systems. We designed a programming language around this, in which all elements of a system (from abstract data types to GUI widgets) are runtime-replaceable components. Our language is called *Dana*[2] and is freely available [1], with a large standard library of components. It is currently used across a range of ongoing projects.

Dana is a multi-threaded imperative programming language, but one that frames these concepts in a component-based structural paradigm. In these terms, Dana has three novel features for our needs that we now present in detail, along with the API that Dana provides to higher layers of RE^X for assembling and perceiving emergent software.

[1] Throughout this paper we use the simplifying assumption that all possible assemblies of an emergent system are valid; in reality an automated unit testing system could potentially provide this validation before particular assemblies are made available for use in the live system.

[2] Dynamic Adaptive Nucleic Architectures: named for its highly dynamic systems of small components with linked internal sub-structures.

```
interface File {
  transfer char path[]
  transfer int pos, mode
  File(char path[], int mode)
  byte[] read(int numBytes)
  int write(byte data[])
  bool eof()
  void close()
}
```

```
component provides App requires File {
  int App:main(AppParam args[]) {
     File ifd = new File(args[0].str, File.READ)
     File ofd = new File(args[1].str, File.WRITE)
     while (!ifd.eof()) ofd.write(ifd.read(128))
     ofd.close()
     ifd.close()
     return 0
  }
}
```

Figure 2 – Example interface to open, read and write files (top); and a component that uses this interface to copy a file (bottom).

2.1.1 Fusing third-party system composition with first-party instantiation

Our first observation is that state-of-the-art realizations of the component-based paradigm are almost completely disjoint from object orientation. Specifically, component models enable third-party instantiation and (re)wiring whereby a so-called *meta-level* controls the composition of a software system by modelling that system as a graph of components (nodes) and wirings (edges). However, within this model they fundamentally lack support for first-party instantiation and reference passing – i.e., the ability to instantiate objects with their own private state and pass references to those objects as parameters. In our experience with existing runtime component models (such as OSGi [3], OpenCom [11] and Fractal [8]), this shortcoming makes it very hard to express many simple modern programming concepts. By contrast, Dana enables first-party instantiation within a paradigm in which a meta-level controls software composition as a graph of components and wirings that can be adapted at runtime.

An example Dana component is shown in the lower half of Fig. 2. This component *provides* an `App` interface and *requires* a `File` interface (defined in the top half of Fig. 2). The component instantiates the `File` interface twice with different parameters, yielding two `File` objects, and copies data from the first file to the second by reading and writing chunks of data. The full system is created by a meta-level which first loads the example component (C_A) into memory, queries its required interfaces, then loads a desired implementing component of `File` into memory and wires C_A's required interface to the respective provided interface of that component. The resulting system is illustrated in Fig. 3. At any point during program execution, the meta-level can choose to rewire the `File` required interface of C_A to point to a corresponding provided interface on a different implementing component, thereby adapting the system's behavior.

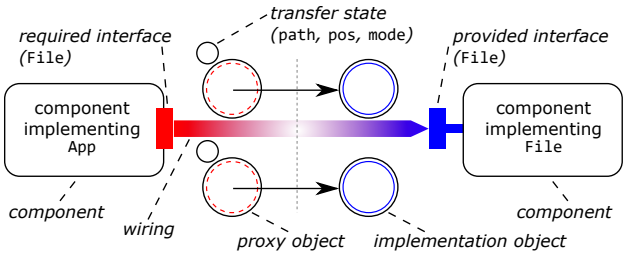

Figure 3 – Internal structure of components and objects. For each instantiated object, a proxy is created with an internal reference to the implementing object from the component of the connected provided interface. Required interfaces maintain a list of all (proxy) objects that were instantiated through them.

Formally, the runtime component model that enables the above is defined as follows. An interface i is a set of *function prototypes*, each comprising a function name, return type and parameter types; and a set of *transfer fields*, typed pieces of state that persist across alternate implementations of the interface during runtime adaptation.

A component c provides one or more such interfaces, where each such provided interface has an implementation scope i_{sc}. An i_{sc} has an implementation of every function of its provided interface (plus other internal functions) and $[0..n]$ global variables (each of which is private, i.e., not visible outside c). A provided interface (and its underlying i_{sc}) of a component must be *instantiated* before use; we refer to these instances as objects. A component c also requires zero or more interfaces, each of which must be connected to a compatible provided interface on another component to satisfy the dependency. These inter-component connections are referred to as wirings.

At an implementation level, a required interface can be thought of simply as a list of function pointers; when a required interface r is wired to a provided interface p, r's function pointers are updated to point at the corresponding functions in the component behind p. On top of this basic mechanism, Dana provides an abstraction of objects such that a required interface can be instantiated many times.

When the code inside a component instantiates one of its required interfaces r (using the language's `new` operator), this results in the instantiation firstly of a special *proxy object* and secondly an i_{sc} instance relative to the provided interface of the component to which r is wired. The i_{sc} has a fresh copy of any (private) global state fields, and of its interface's transfer fields. The proxy object has an internal link to the corresponding i_{sc}. A reference to the proxy object, initially held by the instantiator, can then be passed as a parameter to other functions as desired.

The use of proxy objects allows the implementing component(s) of those objects to be adapted, resulting in a change to the internal links within the proxy objects, without affecting any references to the proxy objects.

2.1.2 A protocol for seamless runtime adaptation

An emergent software system must be able to continually experiment online with different combinations of components without disrupting the system's primary task. To enable this we need a low-overhead runtime adaptation protocol that works with our component model.

We have therefore designed a protocol that uses transparent hot-swapping (similar to that proposed by Soules *et al.* [28]) for zero down-time; and that supports object reference persistence across implementation changes. In addition, it yields fast adaptation for both stateful and stateless components by using a modular protocol design in which different command orderings give different semantics for these two cases, as we explain below. This is much more lightweight than classic 'quiescence'-based approaches for coarse-grained component models, in which potentially large portions of a running system must be deactivated before adaptation proceeds [21, 32].

Algorithm 1 Adaptation protocol

```
1:  srcCom      ▷ Comp. to rewire a required interface of
2:  sinkCom     ▷ Comp. with provided interface to wire to
3:  intfName                ▷ Interface name being adapted
4:  pause(srcCom.intfName)
5:  r_objs = getObjects(srcCom.intfName)
6:  rewire(srcCom.intfName, sinkCom)
7:  resume(srcCom.intfName)
8:  for i = 0 to r_objs.arrayLength − 1 do
9:    if pauseObject(r_objs[i]) then
10:     a = adaptConstruct(sinkCom.intfName, r_objs[i])
11:     b = rewireObject(r_objs[i], a)
12:     resumeObject(r_objs[i])
13:     waitForObject(b)
14:     adaptDestroy(b)
15:   end if
16: end for
```

Pseudocode of our protocol is given in Alg. 1. All adaptation in Dana is performed by changing a component *srcCom*'s required interface r from its current wiring to instead be wired against a compatible provided interface of a different component *sinkCom*.

In simple terms, our two flavours of adaptation proceed as follows. For stateless objects, any new calls on those objects are immediately routed to their new implementation from *sinkCom*, while any existing calls active in the old implementation are (concurrently) allowed to finish.

For stateful objects, any existing calls active in their old implementations are allowed to finish while any new calls are temporarily held at the point of invocation, allowing existing calls to finish potential updates to transfer state fields. When all existing calls finish, all held calls, and any new calls, are then allowed to proceed and are routed to the objects' new implementations from *sinkCom*.

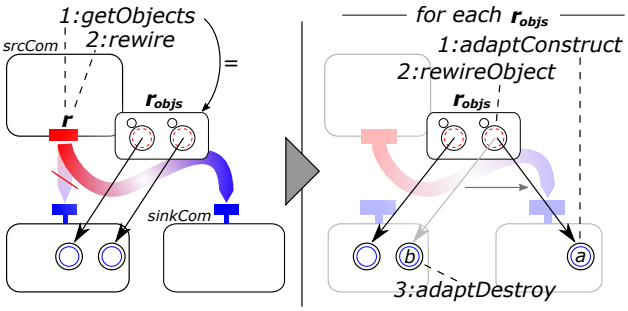

Figure 4 – Adaptation sequence overview. A selected required interface r is rewired, followed by each object in the set r_{objs}.

To achieve these effects, the operations used in our adaptation protocol are defined in detail as follows.

`pause` prevents new objects from being instantiated via r, and prevents any existing instances from being destroyed. Specifically, any Dana language instructions that attempt to instantiate or destroy an object become held at the respective language operator, after checking whether or not r is paused. We call this set of held threads r_{ht}.

`getObjects` acquires a list of all existing objects that have been instantiated via r, giving the list r_{objs}. Because r is currently paused, it is assured that r_{objs} contains *all* objects whose implementations are (and ever will be) sourced from the component to which r is currently wired.

`rewire` changes the current wiring of r to point to the equivalent provided interface of *sinkCom*.

`resume` removes the paused status from r and allows the set of threads in r_{ht} to resume execution, thereby enabling any held object instantiation or destruction operations to proceed. After this point, any instantiation operators will resolve against the component to which r is now wired, rather than to its previous wiring.

Our adaptation protocol uses the above four operations on lines 4–7. The result is that the wiring graph at the component level has been adapted, illustrated in the left half of Fig. 4. The protocol is then left with a set of objects r_{objs} whose implementations belong to the previous wiring of r. To complete the adaptation, each such object must have its implementation updated to be from the current wiring of r. This procedure is performed in the loop from lines 8–16 and is illustrated on the right of Fig. 4.

`pauseObject` first checks if the given object has been destroyed by this point (recall that object destruction was re-enabled by `resume`). If not, this individual object's destruction is prevented by setting a flag on the object such that a destruction operator will be held until the flag is unset; `pauseObject` then returns `true`. The remaining set of operations on lines 10–14 can then proceed in the knowledge that the object they operate on will not be destroyed in the meantime. A successful invocation of `pauseObject` also prevents any new function calls from being made on the given object, holding any such calls at their invocation operator in a set o_{ht}.

`adaptConstruct` dynamically creates a new object from the component to which r is now wired (by `rewire` as described above). This object, a, is specially created in such a way that it shares the transfer state fields (if any exist) of $r_{objs[i]}$, instead of having its own fresh copy.

`rewireObject` changes the internal link of the proxy object to which $r_{objs[i]}$ refers (recall that all references to objects actually refer to their proxy object), pointing that link at a instead of its previous location. As a return value, rewireObject gives a different proxy object b, the internal link of which refers to the object implementation to which the proxy object of $r_{objs[i]}$ used to refer.

`resumeObject` allows any function calls held in o_{ht} to proceed into the object (whose implementation object is now one sourced from *sinkCom*), also allowing all future calls to immediately proceed into the object.

`waitForObject` blocks until all function calls currently operating within the given object complete.

`adaptDestroy` destroys the given object in such a way that its transfer state fields are not also destroyed.

For adaptations of stateless objects, used in cases when the corresponding interface has no state transfer fields, our adaptation protocol is arranged as in Alg. 1. For stateful objects, the `waitForObject` operator instead appears just before line 10 and is given $r_{objs[i]}$ as its parameter. This ensures that the previous implementation has finished any potential modifications to an object's transfer fields before any logic in the new implementation occurs.

2.1.3 Structuring for discoverable code

An emergent software system must be able to autonomously discover usable components for different parts of itself. We wanted to support this without any extra wiring specifications or manifest files, as are common in many component models [19, 16]. In doing so we avoid developers having to do any work beyond writing component functionality. Instead, we chose to make discoverable code an inherent feature of our platform.

Our solution here is simply to define a fixed structure for projects. The root folder of a project therefore contains a 'resources' directory tree containing the source code of all interface types, and a symmetrical directory tree containing all components that implement those interfaces. For a component that declares a required interface of type `io.File`, we then know to look in the directory '`io`' for all potential implementation components of this interface.

2.1.4 Interface to higher system layers

To higher layers of RE^{X}, Dana provides the following API: **load** and **unload** a component into or out of memory; **get** the set of interfaces (provided and required) of a component; **connect** a component's required interface to another component's provided interface (for initial system assembly); and **adapt** a component's required interface to connect to an equivalent provided interface on a different component (via the above adaptation protocol).

2.2 Perception, Assembly and Learning

Whereas Dana provides the fundamental mechanisms to build systems, our perception, assembly and learning framework (PAL) abstracts over entire systems for online learning. Specifically, PAL **assembles** sets of discovered components into working systems; **perceives** the health of those systems and the conditions of their deployment environment; and **learns** correlations between a system's health, its current environment conditions, and its current assembly. Each of these elements operates at runtime, while the target emergent software system is executing. Unlike existing work, we use no models or architectural representations of software [12, 23], instead enabling systems to emerge autonomously as a continuous process.

2.2.1 Assembly

The assembly module of our framework is responsible for discovering the possible units of logic (i.e., components) that can form a given system; assembling a particular configuration of those components to create a working system; and re-assembling the running system to a different configuration using our adaptation protocol.

The assembly module starts with a 'main component' of a target software system (such as that shown in Fig. 2). The required interfaces of this component are read, and all available components that offer compatible provided interfaces are then discovered. This is done using Dana's inherent structuring for discoverable code: the package path of each required interface is converted to a local directory path, which is scanned for any components that provide this interface type. For each component found, the required interfaces of *those* components are read, and further components are discovered with corresponding provided interfaces. This procedure continues recursively until a full set of possible system compositions is discovered, and can be re-run periodically to detect new components.

We expect there to be multiple different implementations of each provided interface because there are typically several ways to solve a given problem, such as the use of different memory cache replacement algorithms or different search algorithms. Each such variant offers equally valid functionality relative to a provided interface, but implementation differences imply that their respective performance characteristics will differ according to different input ranges or deployment environment characteristics that are encountered by the system.

When the discovery procedure is complete, the assembly module provides a list of strings, each of which is a full description of one configuration of components. The assembly module can then be instructed to assemble the target system into one such configuration. If the system is not yet assembled, this means simply loading each component into memory and connecting the appropriate required and provided interfaces together, then calling the main method of the main component to start the system.

If the target system is already assembled in a particular configuration, a command to re-assemble it into a different one uses our adaptation protocol to seamlessly shift to the alternative. In detail, starting from the main component of the target system, the assembly module walks through the inter-component wiring graph to discover the difference points between the current configuration and the new one. For each such difference, the corresponding alternative component is loaded into memory, along with all components (recursively) that it requires, and the adaptation protocol is used to adapt to that component. The old component (and components that it required and that are not in use by other parts of the system) is then removed.

2.2.2 Perception

The ability to assemble and re-assemble a software system into different configurations of components must be guided in some way. Key to this is an ability to perceive the way the software 'feels' at a given point in time, and the way the software's deployment environment 'looks' at correlated points in time. These streams of perception can then be mapped to the software's current assembly (i.e., the way it is behaving) to understand how different behaviors make the software feel in different environments.

This is achieved using our perception module. To enable perception throughout a system we use a `Recorder` interface. Any component can declare a required interface of this type and then use it to report one (or both) of two kinds of data as it sees fit: *events* and *metrics*.

Events represent the way individual software components are perceiving the outside world – their inputs or deployment environment conditions. Events have a standard structure, with a name, descriptor, and value. When an event is reported to a `Recorder`, a timestamp is added.

Metrics represent the way individual software components are perceiving themselves – how they 'feel'. Metrics again have a standard structure, including a name, a value, and a boolean flag indicating whether a high or low value of this metric is desirable. As with events, a timestamp is added to a metric when reported to a `Recorder`.

When a new configuration of the software system is selected via the assembly module, the perception module uses Dana's `getInterfaces` API to check the components of that configuration for any with a `Recorder` required interface. For all such components, their attached `Recorder` implementation component is periodically polled to collect any recently reported events or metrics, noting the component from which they originated.

2.2.3 Learning

Finally, our learning module is tasked with understanding the data from the perception module, and exploring different assembly configurations of the target system to understand how different behavior sets cause the software to react to different external stimuli. We describe the full details of our learning approach in the next section.

2.2.4 Interface to higher system layers

The perception and assembly modules provide the following API to the learning module: **setMain()**, selecting a 'main' component of a program to assemble; **getConfigs()**, returning a list of strings describing every possible configuration of components; **setConfig()**, taking a configuration string to assemble/re-assemble the system to; and **getPerception()**, returning all events and metrics that have been collected since this function was last called.

2.3 Linear bandits for rapid emergence

In this section we describe our approach to efficiently learning the correlations between perception of internal state and external environment, and the currently selected behavior of a system. We first define this problem more precisely with a case study of an emergent web server, and then we describe our learning approach in detail.

2.3.1 Problem definition

For our evaluation in this paper we use a web server as an example emergent software system. A partial structure of this is shown in Fig. 5, illustrating the set of possible configurations that each represent a valid system. For simplicity here we only show components that have variations – in reality, the set of components used to form this system is much larger, at over 30 components (of which only 15 are shown). The components not shown here include those for file system and socket operations, string handling utilities, abstract data type implementations, etc.

From this set of components, there are 42 possible assemblies in total, each of which results in a functional web server system but with differing behaviors. As examples, some such assemblies use a memory cache (of which there are several variants) while others use a compression algorithm; and some use a thread-per-client approach to concurrency while others use a thread-pool approach.

We must then establish which of these 42 options best suits the current external stimuli to which the software is being subjected. These external stimuli may also change, invalidating what has been learned to date and requiring further search iterations. An exhaustive search approach is clearly undesirable, causing the system to spend too long in sub-optimal configurations; we therefore need a way to balance exploration of untested parts of the search space with exploitation of solutions known to be good.

The components of our web server generate two kinds of perception data to inform this. *RequestHandler* implementations report a metric of their average response time to client requests, providing an internal perception of self. Implementations of the *HTTPHandler* interface report events of the resources being requested and their size. This represents the system's perception of its deployment environment. For each set of client request patterns that are input to the system, there then exists one composition of components (behavior) that optimizes the reward value

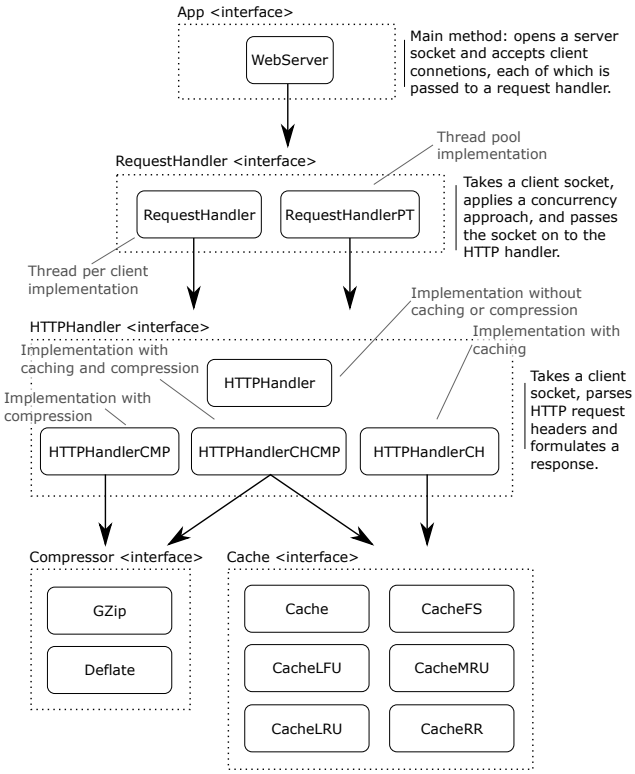

Figure 5 – The set of components from which our web server can emerge. Boxes with dotted lines are interfaces, and those with solid lines are components implementing an interface. Arrows show required interfaces of particular components. The general purpose of each interface's implementations is noted by the interface, and a description of how the available implementation variations of that interface work is also indicated.

(i.e., minimizes response time) for a given environment. In future we expect multiple reward values (from different components) and dimensions of external perception (for example including resource levels of the host machine), but for now the above values are sufficient to explore the concept of runtime emergent software systems.

Our solution to this configuration search and learning problem is based on the statistical learning approach proposed by Scott [27]. In the remainder of this section we present the details of how we apply this approach.

We first cast our fundamental problem as a 'multi-armed bandit', for which the learning approach is intended. We then discuss the concept of Thompson sampling and how we use it to simultaneously update performance estimates of *multiple* configurations after experimenting with just one of them, and how we use Bayesian regression to derive beliefs about individual component performance within a configuration. Besides adapting the approach for our particular problem, we make two changes: (i) we use Bayesian linear regression instead of probit regression, enabling us to handle continuously distributed results; and (ii) we add a simple classifier system to provide memory of environment changes over time.

2.3.2 The Multi-armed Bandit Formulation

Online learning must balance the exploration of undertested configurations with exploiting configurations already known to perform well [27]. The canonical form of this is the multi-armed bandit problem, devised for clinical trials [30, 7], and recently a dominant paradigm for optimization on the web [27, 9]. A multi-armed bandit has a set of available actions called arms. Each time an arm is chosen, a random reward is received, which depends (only) on the selected arm. The objective is to maximize the total reward obtained. While short-term reward is maximized by playing arms currently believed to have high reward, long-term benefit is maximized by exploring to ensure that we don't fail to find the best arm.

In the case of our emergent software, each possible configuration is considered an arm, and the reward given by playing an arm (i.e., selecting a particular configuration of the web server) is defined by our metrics (i.e., the average response time of this configuration to client requests).

One general method for tackling the multi-armed bandit problem is Thompson sampling. Theoretical performance guarantees exist for Thompson sampling in general settings [22, 5, 25], and the technique has been empirically shown to perform extremely well [9, 27]. The key feature of Thompson sampling is that each arm is played with the probability it is the best arm given the information to date. This requires the use of Bayesian inference to produce 'posterior distributions' that code our beliefs about unknown quantities of interest, in this case the expected values of the arms (see Sec. 2.3.3). With this inference, it has been shown that Thompson sampling can be efficiently implemented by drawing a single random sample from the posterior distribution of all unknown quantities, then selecting the arm which performs best conditional on this sampled value being the truth [22].

For example, suppose our unknown quantities are the expected value of each arm, and beliefs about these quantities are encoded as (posterior) probability distributions with densities given by bell curves. The center of the bell curve is then the average reward seen on that arm to date, and the spread is our level of uncertainty (with high uncertainty a result of few observations). For an arm to be selected with Thompson sampling, the random sample from its bell curve must be higher than corresponding samples from all other arms; to have a non-negligible probability of being selected, the distribution must be capable of producing high samples. This is true if either the center point is high or if the spread is large, corresponding respectively to high average observed rewards or high uncertainty.

The effect is that the arms most likely to be played are those that experience suggests are likely to perform well, and those that may perform well but we have insufficient information about. Arms for which we have good information that they will perform badly are played with

very low probability. As more information is gained, and beliefs concentrate on the truth, no arms will remain for which there is insufficient information. Thus, in the long term, optimal arms are played with very high probability.

2.3.3 Forming beliefs

Thompson sampling balances exploration and exploitation in the presence of a Bayesian estimate of arm values. In traditional bandit settings, the value of each arm is estimated independently. However, the combinatorial explosion in the total number of available configurations renders this approach undesirable since each arm will need to be experimented with multiple times. Our emergent software example described above results in 42 such arms; and for example introducing just one further caching variation (see Fig. 5) would increase this to 48. As the complexity of an emergent software system grows, it quickly becomes undesirable to consider all configurations and test each one at runtime. Furthermore, estimating the performance of each configuration independently ignores the fact that many configurations share components, and so are likely to have related performance characteristics.

We therefore follow and use a regression framework based on classical experimental design to *share* information across the different arms available to us [27]. The intuition behind this scheme is that the performance of any configuration using the `HTTPHandlerCH` component is in some way informative for any other configuration involving that component. This is formalized by modelling the expected reward for a given configuration as a function of the components deployed within that configuration. In detail, we code each interface as a factor variable, with number of levels equal to the number of available components for that interface. Standard *dummy coding* is used, so that each level of the factor is compared to a fixed baseline level. For our web server example the effect of this is to model the expected reward of a configuration as

$$\begin{aligned}&\beta_0+\beta_1\mathbb{I}_{\text{RequestPT}}\\&\quad+\beta_2\mathbb{I}_{\text{HttpCMP}}+\beta_3\mathbb{I}_{\text{HttpCH}}+\beta_4\mathbb{I}_{\text{HttpCHCMP}}\\&\quad+\beta_5\mathbb{I}_{\text{Deflate}}\\&\quad+\beta_6\mathbb{I}_{\text{CacheFS}}+\beta_7\mathbb{I}_{\text{CacheLFU}}+\beta_8\mathbb{I}_{\text{CacheMRU}}\\&\quad+\beta_9\mathbb{I}_{\text{CacheLRU}}+\beta_{10}\mathbb{I}_{\text{CacheRR}}\end{aligned}\tag{1}$$

where β_i are unknown real numbers to be estimated, and indicator functions $\mathbb{I}_X$ take value 1 if component X is used in the configuration, and 0 otherwise. Note that coefficients for `RequestHandler`, `HttpHandler`, `GZip` and `Cache` are implicitly coded as baseline levels for the factors, so the above coefficients are interpreted as deviations from this baseline performance. In other words, if for example all of HttpCMP, HttpCH and HttpCHCMP are set to 0, this implies the default `HttpHandler` is in use and its reward is encoded in β_0. The model in Equation 1 can be automatically derived by our learning module, and has only 11 elements to estimate, instead of 42.

The standard linear regression model assumes observed rewards are equal to expected rewards (1) plus a 'noise' term for un-modelled variability. If we denote the vector of binary indicator variables for a given configuration as

$$\begin{aligned}x_{\text{conf}} = (&1, \mathbb{I}_{\text{RequestPT}}, \mathbb{I}_{\text{HttpCMP}}, \mathbb{I}_{\text{HttpCH}}, \mathbb{I}_{\text{HttpCHCMP}},\\&\mathbb{I}_{\text{Deflate}}, \mathbb{I}_{\text{CacheFS}}, \mathbb{I}_{\text{CacheLFU}}, \mathbb{I}_{\text{CacheMRU}},\\&\mathbb{I}_{\text{CacheLRU}}, \mathbb{I}_{\text{CacheRR}}),\end{aligned}$$

[3]

with the vector of unknown coefficients denoted as $\beta = (\beta_0, \beta_1, \ldots, \beta_{10})$, and observed reward as y, the assumed model of linear regression is then that

$$y = x_{\text{conf}}\beta + \varepsilon,$$

where ε is a zero-mean Gaussian random value independent of all other observed quantities, with unknown variance σ^2. After observing multiple configurations and their rewards, we have a list of (x_{conf}, y) pairs; regression then finds the single β value which makes all $x_{\text{conf}}\beta$ values as close as possible to their relative observed y values.

The Bayesian approach to regression is used so that we can support Thompson sampling for action selection; specifically the Bayesian approach produces a posterior probability distribution over β and σ^2 as its output from which to then sample [24]. We use the standard conjugate prior distribution, with σ^2 having an inverse-gamma prior with parameters a_0 and b_0, and β having a multivariate Gaussian prior conditional on σ^2 with parameters $\tilde{\beta}$ and $\sigma^2\Lambda_0^{-1}$. The parameters of the prior are specified in Sec. 2.3.4. The posterior distribution of σ^2 is again an inverse-gamma distribution with parameters updated by the data, and the posterior for β conditional on σ^2 is a multivariate Gaussian distribution dependent on the data.

2.3.4 Implementation

The above approach is implemented in our learning module. This maintains information about the history of selected configurations and the rewards obtained. It also stores an $m \times k$ 'action matrix', where m is the number of valid configurations (in our case 42), and k is the number of unknown regression coefficients β_i (in our case 11). Each row corresponds to a valid configuration, and consists of the vector x_{conf} of indicators for the configuration. Multiplying this action matrix by a vector of coefficients β returns a vector of $x_{\text{conf}}\beta$ values, and thus simultaneously evaluates Equation 1 for all valid configurations.

This action matrix is used when selecting which configuration to deploy. A single β and σ^2 are sampled from the posterior distribution resulting from linear regression and β is then multiplied by the action matrix to get Thompson-sampled values for each arm. The configuration corresponding to the row with the highest resulting value is then chosen and deployed. After a ten second observation window, the resulting reward y is observed, and the (x_{conf}, y) pair is stored. The posterior distribution is then

[3] The initial 1 is included as the intercept term which multiplies β_0 and is present for all configurations.

updated before repeating the process. Pseudocode for this is given in Alg. 2, in which the formulae for sampling from the posterior is given in lines 8–13.

When initializing the system, appropriate values must be chosen for the prior parameters so that the algorithm explores sufficiently without immediately dismissing configurations. We choose $a_0 = 1$ to give a weakly informative prior distribution. b_0 is then chosen so that the range of values supported by the inverse-gamma(a_0, b_0) distribution includes the reward variance in the data; we choose b_0 such that the *a priori* most likely standard deviation, $\sqrt{b_0/2}$, is approximately equal to the expected standard deviation of reward. For β, we use a prior mean $\tilde{\beta}$ with all values except the first equal to zero, as it is unknown how each component affects the performance of the web server. The value of β_0 encodes the base performance of the server; optimistic prior beliefs that β_0 is higher than rewards we actually observe encourages initial exploration as, before a lot of data has been observed, the belief will remain that unexplored configurations have higher rewards than those that have been observed. Thus we take $\tilde{\beta}_0$ (the first component of $\tilde{\beta}$) to be slightly higher than the reward level we actually expect from the system. For Λ_0, the inverse of the prior covariance, we take a default weakly informative prior and set Λ_0 to be the identity matrix multiplied by a small constant value, equal to 0.1 throughout this article. The particular values of $\tilde{\beta}_0$ and b_0 used for our experiments are reported in Sec. 3.

Algorithm 2 Learning Algorithm

```
1:  //matrix of all available x_conf vectors (configurations)
2:  actionMatrix = assembly.getConfigs()
3:  X = new Matrix() //list of observed x_conf's to date
4:  y = new Vector() //list of rewards seen for each X
5:  n = 0
6:  while running do
7:    //do linear regression & sample from posterior
8:    Λ = X^T X + Λ_0
9:    β = Λ^{-1}(Λ_0 β̃ + X^T y)
10:   a = a_0 + (n/2)
11:   b = b_0 + (y^T y + β̃^T Λ_0 β̃ − β^T Λ β) × 0.5
12:   σ^2 = new InverseGamma(a, b).sample()
13:   sample = new Normal(β, σ^2 Λ^{-1}).sample()
14:
15:   //select the new configuration to use
16:   i = arg max(actionMatrix * sample)
17:   assembly.setConfig(i)
18:
19:   //wait for 10 seconds, then record observations
20:   result = 1/perception.getAverageMetric()
21:   add row i of actionMatrix as new row of X
22:   add result as new element of y
23:   n++
24: end while
```

2.3.5 Handling deployment environment changes

In a traditional multi-armed bandit problem the reward distributions of each arm, while unknown to the player, do not change their distribution over time. Thus, if the optimal arm is found, playing that arm forever carries no disadvantage. In a software system, however, the rewards of the respective arms (i.e., configurations) may change over time as the deployment environment of the system changes. In our example, if the request pattern experienced by the web server changes, then the effectiveness of a given configuration may diverge from current estimates. Without accommodating for this, when the request pattern changes the system must take time to first 'unlearn' what it knows about the effectiveness of the available configurations and their constituent components, and then learn new estimates. If the request pattern then reverts back to its old form, the entire procedure must be repeated.

To optimize this, we augment our algorithm with the ability to categorize request pattern features, and to update its estimates accordingly. However, automatically deriving such categorizations in real-time is itself a challenging problem. For this paper we manually define two features, based on how we presume they will affect the web server.

The first feature is *entropy*, describing the number of different resources requested in a given time frame. High entropy indicates many different resources, while zero entropy indicates a single resource requested repeatedly. A pattern with low entropy, where many requests are the same, may benefit from configurations using a caching component, while for high entropy patterns caching would not help, and may even be detrimental.

The second feature is *text volume*, describing how much of the content requested in a given time frame was textual (i.e., HTML, CSS or other text-based content). A request pattern with high text content will likely be served better by a configuration that makes use of a compression component, as text is highly compressible, whereas a request pattern with high image or video content would waste resources by using compression and achieve little as a result.

We have implemented a simple pattern-matching module that observes the stream of events from our perception module and classifies them as follows: if one type of request (video, text, or image) makes up more than half of the requests in an observation window, it is assumed the request pattern has 'low' entropy, and otherwise 'high'. If more than half of the requests made in an observation window are for text items, it is assumed that the request pattern currently is 'high' text, otherwise 'low'.

To incorporate these environment features in our learning approach, we add terms to Equation 1 corresponding to these features, and also interaction terms between environmental indicators and components we believe to be relevant. In particular, we expect text volume to affect the

benefit of compression, and entropy to affect the benefit of caching. Equation 1 is therefore modified to consist of the following indicators, each with a regression coefficient β_i:

$$(1, \mathbb{I}_{\text{RequestPT}}, \mathbb{I}_{\text{HiEnt}}, \mathbb{I}_{\text{HiTxt}}, \mathbb{I}_{\text{HttpCMP(LowTxt)}}, \mathbb{I}_{\text{HttpCMP(HiTxt)}}, \mathbb{I}_{\text{HttpCH(LowEnt)}}, \mathbb{I}_{\text{HttpCH(HiEnt)}}, \mathbb{I}_{\text{HttpCHCMP(LowTxt,LowEnt)}}, \mathbb{I}_{\text{HttpCHCMP(HiTxt,LowEnt)}}, \mathbb{I}_{\text{HttpCHCMP(LowTxt,HiEnt)}}, \mathbb{I}_{\text{HttpCHCMP(HiTxt,HiEnt)}}, \mathbb{I}_{\text{Deflate}}, \mathbb{I}_{\text{CacheFS}}, \mathbb{I}_{\text{CacheLFU}}, \mathbb{I}_{\text{CacheMRU}}, \mathbb{I}_{\text{CacheLRU}}, \mathbb{I}_{\text{CacheRR}}). \quad (2)$$

Adding the environment indicators, and splitting the indicators for different HTTP handlers by the environment, adds 7 extra regression coefficients. It also increases the number of possible 'configurations' to $42 \times 4 = 168$, as each configuration can now be observed in 4 environment states. After a configuration is deployed, the resulting vector x_{conf} of an observation window includes the environment indicators and interaction indicators (i.e., all the indicators in Equation 2). The linear regression proceeds as before, but with k increased so that we still have one regression coefficient per indicator (in this case $k = 18$).

When it is time to select an action, we sample a β value from our posterior distribution as before, and multiply the action matrix by β to give the predicted value. However, not all configurations are available to us, since some are determined by the environment. We make the simplifying assumption that the environmental context in the current time period will be the same as in the previous period, and restrict our configuration to those that correspond to that environment. It is plausible that a model of the evolution of the environment could be built and used to improve the prediction of values, but is beyond the scope of this paper.

The effect of this enhancement is that components' performance levels in different environments may be updated without having to forget information when the environment changes. We see the benefit of this in Sec. 3.3.

3 Experimental Evaluation

The goal of our evaluation is to investigate whether optimal designs of a software system emerge rapidly using real-time learning. Specifically, we evaluate our approach in three key ways. We first examine the speed with which runtime adaptation occurs. This helps to show the viability of emergent software at runtime, which may frequently adapt in exploration periods. Second, we manually analyze the different possible compositions of our web server as a baseline, demonstrating that different optimals exist in different operating environments as a result of micro-variation. This validates our emergent software approach. Third, we examine RE$^{\text{X}}$ in operation, particularly the effectiveness of our online learning approach to discover optimal compositions of behavior in real time.

Our evaluation is conducted using a real, live implementation of the emergent web server described in Sec. 2.3.1, orchestrated by RE$^{\text{X}}$. We run our system on commodity rackmount servers, hosted in a production datacenter, of a similar design to many datacenters around the world. In particular we used servers with Intel Xeon Quad Core 3.60 GHz CPUs and 16 GB of RAM, running Ubuntu Server 14.04. Similar machines were used as clients when generating workloads for our system, where client machines were situated on a different subnet (in a different physical building) to the server machines.

All of our source code, with instructions on how to reproduce all results reported here, is available online at [4].

	Average	Maximum	Minimum
setConfig (idle)	509.60 ms	615.00 ms	397.00 ms
setConfig (busy)	1350.32 ms	5811.00 ms	510.00 ms
pause/resume (idle)	8.50 µs	9.94 µs	7.81 µs
pause/resume (busy)	13.22 µs	31.21 µs	8.51 µs
pauseObject/resumeObject (idle)	4.51 µs	5.34 µs	3.84 µs
pauseObject/resumeObject (busy)	28.54 µs	387.17 µs	4.35 µs
components adapted in setConfig()	1.22	3.00	1.00

Table 1 – Adaptation speed measured in different ways, from full configuration changes to individual component adaptations.

3.1 Adaptation characteristics

We use our highly adaptive Dana programming language (see Sec. 2.1) to support low-cost adaptation. This is a key enabler of emergent software systems, which must be able to experiment with various configurations and adapt to those configurations when appropriate during program execution. In this section we evaluate the time taken to perform runtime adaptation in detail.

We consider two factors in performing runtime adaptation: the overall time taken by RE$^{\text{X}}$ to move from one complete configuration to another; and the time taken to perform a single adaptation between two components. For each test we perform 100 configuration changes (moving to each of our 42 configurations at least twice) with a 5 second gap between each configuration change. Across all tests we assume that any components needed are already loaded into memory and ready for use.

The first of the factors we consider, moving from one complete configuration to another, involves parsing a configuration string passed to `setConfig()`, verifying the validity of a configuration, and performing the staged adaptation procedure for each point at which the new configuration differs from the current one. The first two rows in Table 1 show the average, maximum and minimum time taken to do this across 100 tests. The first row, marked 'idle', shows results when the web server is given no workload, while the second row marked 'busy' shows results when the web server is given a workload that causes it to use 100% CPU capacity. This indicates that the use of `setConfig()` is generally slower when the

web server is busy. There are two reasons for this: first, `setConfig()` is processed on the same physical machine as the system under its control and so is given less CPU time when that system is busy; and second, when more requests are in progress at the web server, the adaptation protocol must wait longer for in-progress cross-component calls to finish (i.e., at `waitForObject`).

We now examine the time taken to perform a single adaptation between two components, using the adaptation protocol described in Sec. 2.1.2. This reveals the time that a part of the web server is actually paused and will therefore delay performance of one or more of its tasks. This can happen for two reasons: the `pause` operation on a required interface, which temporarily prevents new objects from being instantiated until `resume` is called, or the `pauseObject` operation on a particular object, which temporarily prevents any new function calls being made into that object until `resumeObject` is called. For these results we first note the difference in time unit compared to the above: pause durations are on the order of *microseconds*. With an idle web server, pause durations are higher across `pause`/`resume` as the base complexity of these instructions is higher (in particular building the object list r_{objs}). Under load, however, pause durations are dominated by `pauseObject`/`resumeObject`, as the list of held inter-object threads o_{ht} grows quickly and must then be iterated over to release each one (see Sec. 2.1.2).

The final row in Table 1 shows the average, maximum and minimum number of adaptations made during one `setConfig()` operation, indicating how many of the above microsecond pauses will occur for our web server during a configuration change. This is very low in our system, with a maximum of just three adaptations. This indicates that system configuration changes during emergent software exploration – which occur at 10 second intervals under our learning algorithm – will be of low impact.

3.2 Manual analysis of divergent optimality

Our approach to emergent software uses small software components, with differing implementations of the same features such as varied cache replacement algorithms, to enable optimal software to emerge by trying differing combinations of these components at runtime.

In this section we validate this approach, in particular showing that there are different configurations of our web server with different performance profiles under different operating environment ranges – necessitating the need to switch between them in order to maintain optimality over time. We refer to this property as *divergent optimality*. To understand whether or not divergent optimal configurations exist for our web server, we run every possible configuration against various client workload patterns. We then examine the resulting performance of each configuration, measured at the server as the time between receiving a request and sending the last byte of the response.

Request pattern	File size (b) [GZ]	Default	Caching	Caching & compression
Text *low entropy*	156,983 [12,757]	11.94 ms	9.56 ms	**0.70 ms**
Text *low entropy*	82,628 [11,949]	4.05 ms	**0.60 ms**	0.66 ms
Text *low entropy*	3,869 [1,930]	1.18 ms	**0.59 ms**	0.63 ms
Image *low entropy*	1,671,167 [1,667,464]	160.81 ms	**150.72 ms**	154.42 ms
Image *low entropy*	84,760 [66,914]	4.02 ms	**0.66 ms**	0.74 ms
Image *low entropy*	4,001 [3,895]	1.22 ms	**0.55 ms**	0.62 ms
Text *high entropy*	156,983 [12,757]	19.27 ms	19.66 ms	**3.04 ms**
Text *high entropy*	82,628 [11,949]	4.61 ms	3.27 ms	**3.07 ms**
Text *high entropy*	3,869 [1,930]	**1.25 ms**	2.93 ms	2.52 ms
Image *high entropy*	1,671,167 [1,667,464]	**156.50 ms**	156.64 ms	157.66 ms
Image *high entropy*	84,760 [66,914]	4.48 ms	3.19 ms	**2.94 ms**
Image *high entropy*	4,001 [3,895]	**1.30 ms**	2.90 ms	2.67 ms

Table 2 – Results of different configurations under different request patterns, showing average response times. The standard deviation throughout these results is low, at around 0.2.

Our results are shown in Table 2, which lists the fastest configuration from each group of configurations (i.e., the fastest configuration that uses neither caching nor compression, the fastest that uses caching, and the fastest that uses both caching and compression). We do not show results for configurations that only use compression, as they reliably perform worst across all of our experiments.

First we subject all configurations to client request patterns with low entropy. We divide this into two subcategories: text-dominated and image-dominated. The results are shown in the top half of Table 2, which also shows the general size of the files being requested along with the compressed size of these files using the GZip algorithm. For almost all of these low entropy request patterns, configurations with caching perform best – marginally better than those with both caching and compression. On investigation, this is because our configurations that use both compression and caching actually use slightly more instructions to check if a compressed version of a file is in the cache (i.e., they append '.gz' to the resource name before checking if that resource is in the cache). In most cases this slight delay is larger than the added network delay of sending the uncompressed version of the file. When the compression win is big enough, however, as in the first row of Table 2, the reverse is true and the caching plus compression solution is faster.

Next we subject all configurations to client request patterns with high entropy; specifically patterns that cycle through a set of 20 popular files. The corresponding results are shown in the lower half of Table 2, again divided into text-dominated and image-dominated requests. Here we see a different picture: in half of the tests, configu-

rations without caching or compression are fastest. This is because, in these cases, reading from the local disk is very slightly faster than searching the cache. On investigation, in cache implementations that use a hash table this is caused by collisions in the hash function which then result in a linear search of a hash table bucket, adding latency. However, this result is reversed for the other half of these results, when the average compression ratio of the files being requested is sufficiently high that the network bandwidth saving overtakes the cache search latency.

These results confirm there are different optimal configurations of components that can form our target system in different environments. Low entropy and high text conditions favor configurations with caching and compression; low entropy and low text conditions favor configurations with caching only; and high entropy conditions favor a mixture of configurations. The subtleties within these results, and the fact that issues such as disk/memory latency will vary across machines, further motivate a real-time, machine-learning-based solution to building software.

3.3 Learning evaluation

We now examine the efficacy of RE$^{\text{X}}$ at discovering the above results, on a live system, starting from no information. While some of these results may be obvious to a human observer, we provide the first example of an autonomous system able to rapidly assemble a corresponding solution at runtime. Our learning system uses only events and metrics reported by components of the live web server (Sec. 2.3.1), alongside the ability to dynamically assemble different configurations of discovered components, to find the best course of action over time.

By default our learning approach tries to maximize $1/responseTime$. Accordingly, we configure our learning algorithm with $\tilde{\beta}_0 = 1$, which is larger than the reciprocal of the response times in Table 2. The standard deviations in this data are on the order of 0.2, and we thus set $b_0 = 0.1$ so these are on a similar scale to $\sqrt{b_0/2}$.

As a theoretical baseline learning comparison, consider an approach that tests each configuration once before selecting the one that performed best. This takes 42 testing iterations before there is a chance of reaching optimality (as there are 42 available configurations), even without any noise in the observations whereby a configuration may need to be tested multiple times. Assuming each such test takes 10 seconds to get an average response time, this means a total of 420 seconds (7 minutes) to reach optimality. If successful, our approach should perform significantly better than this baseline in most cases.

We use a range of request patterns to evaluate our emergent software system, starting with simpler patterns. Each experiment is repeated 1,000 times and the interquartile ranges plotted. Each graph is plotted as *regret*, which is calculated as $(1/responseTime)_{chosenAction} - (1/responseTime)_{optimalAction}$ for each point in time

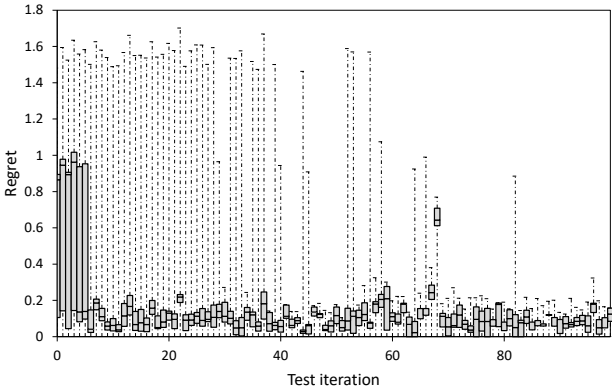

Figure 6 – Learning using response times to small text files.

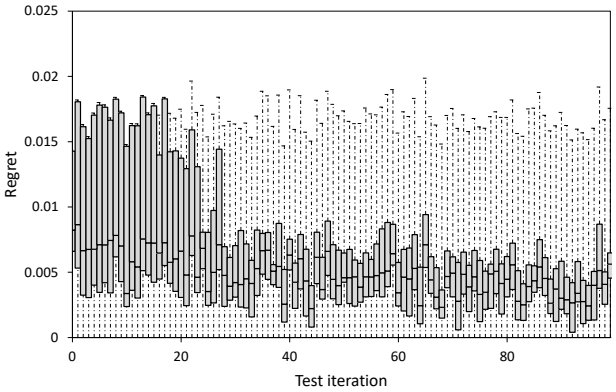

Figure 7 – Learning using response times to large text files, with adjusted prior values for $\tilde{\beta}_0$ and b_0.

(where knowledge of the optimal actions over time is based on our manual analysis of these request patterns). Specifically, the shaded boxes on these graphs show the size of the interquartile range from the distribution of regret results at each time step across all 1,000 experiments. The horizontal line dividing each shaded box is the median value. The whiskers above / below the shaded boxes show the highest / lowest results across all experiments.

Fig. 6 shows results for request patterns of small HTML files with low entropy. Here we see a dramatic reduction in regret after only a few iterations (where one 'iteration' represents a 10-second observation window). Although high regret is occasionally seen after this point, this is an inevitable artefact of continual exploration. Very good response times are learned here after just 50 seconds, which is significantly faster than the baseline described above. This demonstrates that our learning approach, based on estimating individual component contribution and then sharing information across all potential configurations, is very effective at avoiding exhaustive experimentation.

In Fig. 7 we show results for request patterns of large HTML files with low entropy. Here we see that the scale of our rewards has changed – i.e., the average response time for larger files is higher (almost 10 times) and as such our prior parameters were observed not to match

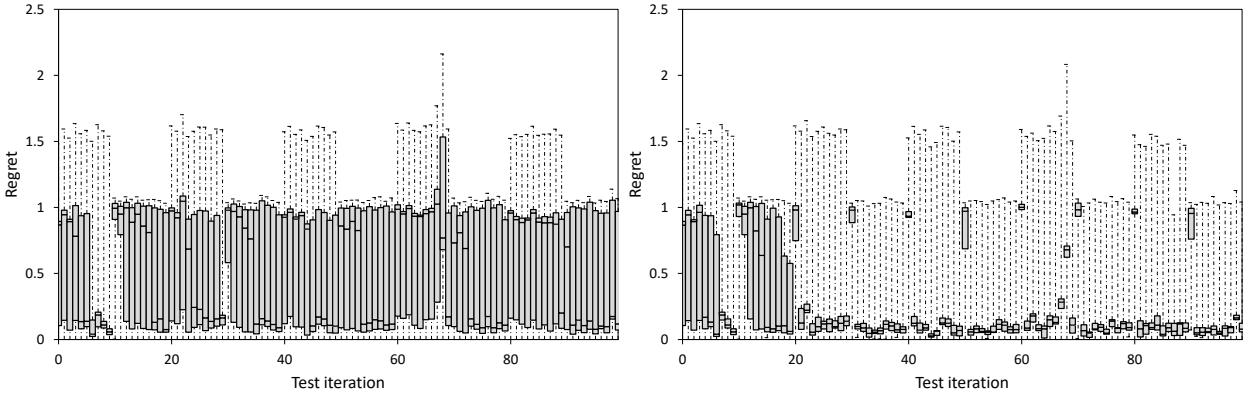

Figure 8 – Learning without (left) and with (right) categorization on a request pattern that changes every ten iterations.

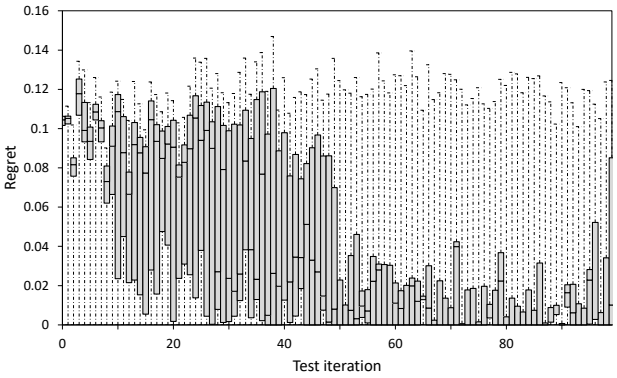

Figure 9 – Learning using response times to a realistic (and highly varying) request pattern, using the NASA server trace [2].

the data. For this experiment we therefore used adjusted prior parameters $\tilde{\beta}_0$ and b_0 that were each divided by 10 compared to the previous test. Again we see rapid convergence on an optimal software assembly, this time at around 20 iterations of the learning algorithm (roughly 200 seconds). The longer convergence time here is due to there being fewer samples from which to draw information (i.e., serving each request takes longer, providing less data per observation time window). Note that good prior values can easily be chosen automatically by sampling response times and calculating their mean and variance.

In Fig. 9 we show results from a real-world web server workload of highly mixed resource requests, taken from the publicly available NASA server trace [2]. This is a challenging request pattern due to its high variance over time, in which different kinds of resource are requested in very different volumes. As a result our learning approach finds it more difficult to compare like-for-like results as different configurations are tested. Initially regret here is generally high, but decreases steadily up to the 40th iteration mark. Overall the system still shows increased performance at least as well as our baseline.

Finally, we examine situations in which request patterns change between different characteristics of entropy and text volume, showing the ability of our platform to adjust to new external stimuli and remember historical information. This is demonstrated in Fig. 8, showing the results of tests in which the request pattern is alternated every ten iterations. When the system operates without categorization, shown on the left of Fig. 8, there is no clear change in regret as it must constantly 'forget' and 're-learn' estimates due to the shifting performance of configurations that it observes. However, with categorization the system exhibits learning behavior for the first two changes in request pattern, and then consistently makes low-regret choices despite the alternation between patterns. There is a brief increase in regret each time the request pattern changes, caused by the learning algorithm needing one observation window in which to observe the changes. This demonstrates that our addition of a simple pattern-matching system achieves the desired effect.

Overall, our results show rapid convergence on optimal software which emerge from online experimentation with different available configurations, with very little information about the nature of the target software system and the deployment conditions that it may experience. RE$^{\mathrm{X}}$ can be deployed on any hardware configuration (which may change the effects seen in Sec. 3.2), and in any deployment environment conditions, and will continually find the most effective system design. More broadly, RE$^{\mathrm{X}}$ can also show the rationale behind its choices to human developers, potentially leading to new development directions.

4 Related Work

While autonomic, self-adaptive and self-organizing computing are now well established, there is relatively little work at the level of autonomous runtime software composition (compared to a much larger body of work on autonomous parametric tuning). The majority of this work is model-driven – relying on substantial human-specification or offline training cycles, or using simple online heuristic search algorithms over carefully specified models. We survey the most closely related work below.

Grace *et al.* propose the use of human-specified adaptation policies to select between different communication interfaces in a river-monitoring scenario [17]. While the use of such adaptation policies is viable in simpler architectures, this becomes infeasible in more complex configurations where the set of component interactions is much larger. We therefore use an online learning approach to effectively discover the adaptation policy at runtime.

Chen *et al.* propose a weighted decision graph of service levels to generate model transformations in an online shopping system [10]. Wang *et al.*, meanwhile, propose a framework that exploits variability of software configurations for self-repair, using a goal model based on formal requirements [33]. By contrast we use a model-free approach in which components report their own current status from which we then infer global properties.

Bencomo *et al.* propose dynamic decision networks (a form of state machine), alongside a models-at-runtime approach to software composition, to decide at runtime between different network topologies for a remote data mirroring system based on perceived resilience levels [6]. This requires pre-specification of the decision network to determine configuration selection, rather than the online learning approach we take for emergent software.

Kouchnarenko and Weber propose the use of temporally-dependent logic to control software configuration, with a domain-specific notation to model temporal dependencies between adaptation actions in a self-driving vehicle control system [20]. While such temporal models may be a useful addition to constrain adaptation, they are again specified by human developers at design time rather than learned at runtime.

In FUSION [12], a feature-model framework is presented that uses offline training combined with online tuning to activate and deactivate selected feature modules at runtime (such as security or logging). Dynamic Software Product Lines [18] generalize the feature model approach as part of the software development process, typically using a pre-specified set of rules to trigger feature activation / deactivation at runtime. Our approach does not use a feature model, instead emerging a working system from a pool of components using online learning.

In SASSY [23], a self-adaptive architecture framework for service-oriented software is presented, using a set of models to describe software architecture and its QoS traits. Further work by Ewing and Menascé [13] applies a set of runtime heuristic search algorithms to the configuration search problem, including hill climbing and genetic algorithms. Our work differs in two ways: first we use a model-free approach, in which system composition is autonomously driven from a 'main' component; and second we apply a statistical machine learning approach to configuration search, based on sharing inferred per-component performance data across configurations.

Finally, we note that Thompson sampling with regression was first proposed by Scott [27] to select likely high performing versions of websites (i.e., with high vs. low quality images), updating beliefs on similar versions without needing to try each individually. We have applied this concept to runtime emergent software, but using Bayesian linear regression (rather than probit regression) to handle continuously distributed results, and a simple pattern matching approach to account for distinct workload patterns that cause different optimal software configurations.

5 Conclusion

Current approaches to self-adaptive software architectures require significant expertise in building models, policies and processes to define how and when software should adapt to its environment. We have presented a novel approach to runtime emergent software which avoids all such expertise, using purely machine-driven decisions about the assembly and adaptation of software. The result is to almost entirely remove human involvement in how self-adaptive systems behave, making this machine-led; and to produce systems that are responsive to the actual conditions that they encounter at runtime, and the way they perceive their behavior in these conditions.

Our approach has three major contributions that form our RE^X platform: a programming language for highly-adaptive assemblies of behaviors; a perception, assembly and learning framework to discover, monitor and control available assemblies; and a learning approach based on linear bandits that solves the resulting search space explosion by sharing information across assemblies.

Our results show that our approach is highly effective at rapidly discovering optimal compositions of behavior in a web server example, balancing exploration with exploitation, and is also highly responsive to changes in the software's deployment environment conditions over time.

In our future work we will broaden our approach to other types of application, and will also explore the automated generation of component variants, and further automation in environment classification. In the longer term we will continue to work towards shifting the system design paradigm even further into software itself – making software a leading member of its own development team.

Acknowledgements

This work was partially supported by the EPSRC *Deep Online Cognition* project, grant number EP/M029603/1. Roberto Rodrigues Filho would like to thank his sponsor, CAPES Brazil, for scholarship grant BEX 13292/13-7. The feedback of Prof. Gordon Blair and Dr. Sarah Clinch was very helpful in preparing this paper for submission. We also thank all of our reviewers for their thoughtful comments which helped to further improve the published version, and particularly our shepherd Dr. Petros Maniatis for his detailed feedback across several final iterations.

References

[1] Dana language: http://www.projectdana.com/.

[2] NASA web server trace: http://ita.ee.lbl.gov/html/contrib/NASA-HTTP.html.

[3] OSGI alliance: https://www.osgi.org/.

[4] Source code and experiments from this paper: http://research.projectdana.com/osdi2016porter.

[5] S. Agrawal and N. Goyal. Thompson sampling for contextual bandits with linear payoffs. *arXiv preprint arXiv:1209.3352*, 2012.

[6] N. Bencomo, A. Belaggoun, and V. Issarny. Dynamic decision networks for decision-making in self-adaptive systems: A case study. In *Software Engineering for Adaptive and Self-Managing Systems (SEAMS), 2013 ICSE Workshop on*, pages 113–122, May 2013.

[7] D. A. Berry and B. Fristedt. *Bandit problems: sequential allocation of experiments (Monographs on statistics and applied probability)*. Springer, 1985.

[8] E. Bruneton, T. Coupaye, M. Leclercq, V. Quema, and J.-B. Stefani. An open component model and its support in java. In *Component-Based Software Engineering*, volume 3054 of *LNCS*, pages 7–22. Springer Berlin Heidelberg, 2004.

[9] O. Chapelle and L. Li. An empirical evaluation of thompson sampling. In *Advances in neural information processing systems*, pages 2249–2257, 2011.

[10] B. Chen, X. Peng, Y. Yu, B. Nuseibeh, and W. Zhao. Self-adaptation through incremental generative model transformations at runtime. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 676–687, New York, NY, USA, 2014. ACM.

[11] G. Coulson, G. Blair, P. Grace, F. Taiani, A. Joolia, K. Lee, J. Ueyama, and T. Sivaharan. A generic component model for building systems software. *ACM Trans. on Comp. Systems*, 26(1):1:1–1:42, Mar. 2008.

[12] A. Elkhodary, N. Esfahani, and S. Malek. Fusion: A framework for engineering self-tuning self-adaptive software systems. In *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE '10, pages 7–16, New York, NY, USA, 2010. ACM.

[13] J. M. Ewing and D. A. Menascé. A meta-controller method for improving run-time self-architecting in SOA systems. In *Proceedings of the 5th ACM/SPEC International Conference on Performance Engineering*, ICPE '14, pages 173–184, New York, NY, USA, 2014. ACM.

[14] F. Faniyi, P. R. Lewis, R. Bahsoon, and X. Yao. Architecting self-aware software systems. In *Software Architecture (WICSA), 2014 IEEE/IFIP Conference on*, pages 91–94, April 2014.

[15] A. G. Ganek and T. A. Corbi. The dawning of the autonomic computing era. *IBM Syst. J.*, 42(1):5–18, Jan. 2003.

[16] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler. The nesc language: A holistic approach to networked embedded systems. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, PLDI '03, pages 1–11, New York, NY, USA, 2003. ACM.

[17] P. Grace, D. Hughes, B. Porter, G. Blair, G. Coulson, and F. Taiani. Experiences with open overlays: a middleware approach to network heterogeneity. In *Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008*, pages 123–136, April 2008.

[18] M. Hinchey, S. Park, and K. Schmid. Building dynamic software product lines. *Computer*, 45(10):22–26, Oct 2012.

[19] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. The click modular router. *ACM Trans. Comput. Syst.*, 18(3):263–297, Aug. 2000.

[20] O. Kouchnarenko and J.-F. Weber. Adapting component-based systems at runtime via policies with temporal patterns. In *Formal Aspects of Component Software*, pages 234–253. Springer, 2014.

[21] J. Kramer and J. Magee. The evolving philosophers problem: dynamic change management. *IEEE Transactions on Software Engineering*, 16(11):1293–1306, 1990.

[22] B. C. May, N. Korda, A. Lee, and D. S. Leslie. Optimistic bayesian sampling in contextual-bandit problems. *The Journal of Machine Learning Research*, 13(1):2069–2106, 2012.

[23] D. Menascé, H. Gomaa, S. Malek, and J. Sousa. SASSY: A Framework for Self-Architecting Service-Oriented Systems. *Software, IEEE*, 28(6):78–85, Nov 2011.

[24] A. O'Hagan. *Kendall's Advanced Theory of Statistics: Bayesian inference. vol. 2B*. Number v. 2, pt. 2 in Kendall's library of statistics. Edward Arnold, 1994.

[25] D. Russo and B. Van Roy. Learning to optimize via posterior sampling. *Mathematics of Operations Research*, 39(4):1221–1243, 2014.

[26] M. Salehie and L. Tahvildari. Self-adaptive software: Landscape and research challenges. *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, 4(2):14, 2009.

[27] S. L. Scott. A modern bayesian look at the multi-armed bandit. *Applied Stochastic Models in Business and Industry*, 26(6):639–658, 2010.

[28] C. Soules, J. Appavoo, K. Hui, R. Wisniewski, D. Da Silva, G. Ganger, O. Krieger, M. Stumm, M. Auslander, M. Ostrowski, B. Rosenburg, and J. Xenidis. System support for online reconfiguration. In *Proceedings of the USENIX Annual Technical Conference*, pages 141–154, June 2003.

[29] C. Szyperski, D. Gruntz, and S. Murer. *Component Software: Beyond Object-Oriented Programming*. Acm Press Series. ACM Press, 2002.

[30] W. R. Thompson. On the Likelihood that one unknown probability exceeds another in view of the evidence of two samples. *Biometrika*, 25(3-4):285–294, 1933.

[31] S. Tomforde, J. Hähner, and C. Müller-Schloer. Incremental design of organic computing systems - moving system design from design-time to runtime. In *Proceedings of the 10th International Conference on Informatics in Control, Automation and Robotics*, pages 185–192, 2013.

[32] Y. Vandewoude, P. Ebraert, Y. Berbers, and T. D'Hondt. Tranquility: A low disruptive alternative to quiescence for ensuring safe dynamic updates. *IEEE Transactions on Software Engineering*, 33(12):856–868, 2007.

[33] Y. Wang and J. Mylopoulos. Self-repair through reconfiguration: A requirements engineering approach. In *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, pages 257–268. IEEE Computer Society, 2009.