March 2017 COMPUTER SCIENCE APPLIED Issue # 4

THE MORNING PAPER QUARTERLY REVIEW

TOWARDS DEEP SYMBOLIC REINFORCEMENT LEARNING

LOAD TESTING IN PRODUCTION AT FACEBOOK

INFERRING A MOBILE PHONE PASSWORD VIA WIFI SIGNALS

A selection of papers from the world of computer science, as featured by Adrian Colyer on The Morning Paper blog

InfoQ queue

CONTENTS

Issue # 4, March 2017

WELCOME...

Welcome to the fourth edition of The Morning Paper Quarterly Review, and thanks once more to the wonderful folks at InfoQ for putting it together. I covered over 50 papers on The Morning Paper during the fourth quarter of last year, so whittling that down to just five choices was tough! I hope you find something that sparks your imagination here. If you find yourself wanting more, there's a fresh paper write-up every weekday on The Morning Paper blog at https://blog.acolyer.org.

Thanks for reading,
Adrian Colyer, Venture Partner, Accel.

Adrian Colyer

SIMPLE TESTING CAN PREVENT MOST CRITICAL FAILURES

Yuan et al. 2014

I first heard about this paper through Caitie McCaffrey, and it's a real gem with information you can put to use straight away! The authors analyse 198 user-reported failures from real-world deployments of distributed systems and provide some fascinating insights into what caused those failures. Nearly every _catastrophic_ failure turned out to be a result of incorrect handling of _non-fatal_ errors - the untested (or even not implemented at all!) exception handling logic caused a cascade of failures once invoked. The good news is that simply by focusing on the exception paths during your code reviews, you can save yourself a lot of potentially damaging downtime down the line. We also learn that even in complex distributed systems, unit tests will catch almost 80% of all failures, and a three node deployment is enough to reproduce 98%. The static checking tool that the authors wrote about found over 500 bugs and bad practices across nine widely used systems; who knows what something similar might find in your own code base?

TOWARDS DEEP SYMBOLIC REINFORCEMENT LEARNING

Garnelo et al. 2016

This was a real wow paper for me when I first read it. Deep learning is definitely top of the tree right now, but before machine learning and deep learning rose to prominence, symbolic reasoning systems were also heavily researched. Deep symbolic reinforcement learning offers a tantalising glimpse of how we might be able to have the best of both worlds: a deep learning system that learns effective symbolic representations, and then a symbolic reasoning system on top for action selection. Compared to a Deep Q Network, a Deep Symbolic Reinforcement Learning system showed much better generalisation to new task variations. The marriage opens up enticing possibilities to use more of the classical AI techniques in conjunction with modern machine learning - for example, inductive logic, analogical reasoning, and planning.

WHEN CSI MEETS PUBLIC WIFI: INFERRING YOUR MOBILE PHONE PASSWORD VIA WIFI SIGNALS

Li et al. 2016

My Twitter feed went nuts after I published the post on this paper. By carefully setting up a public wifi hotspot (in a café for example), it's possible to infer the digits entered by a user on their phone when typing in a PIN... just by looking at the interference in the wifi signal strength caused by the movements of the user's hand! It's a really clever side-channel attack that both amazes for its ingenuity, and at the same time leaves you wondering if there's _anything_ in this world that is truly safe from determined attackers!

KRAKEN: LEVERAGING LIVE TRAFFIC TESTS TO IDENTIFY AND RESOLVE RESOURCE UTILISATION BOTTLENECKS IN LARGE SCALE WEB SERVICE

Veeraraghavan et al. 2016

Kraken is a load testing system at Facebook. What's really interesting about Kraken though, is that instead of trying to emulate real workload traffic in a test system, Kraken uses real user traffic in production systems! It's a genius solution to a hard problem. Kraken works by updating the routing layer to direct a greater proportion of the overall traffic to the systems under test. In other words, Facebook deliberately and routinely stress live systems using live traffic. That sounds kind of scary, in the same way that Chaos Monkey sounds scary when you first encounter the idea. But gradually Facebook's teams came to embrace Kraken and look forward to the tests as a way of better understanding their systems. The paper lists a whole bunch of benefits that came out of regularly running Kraken tests, including a 20% increase in request serving capacity as a result of the improvements made to their systems from the findings.

MORPHEUS: TOWARDS AUTOMATED SLOS FOR ENTERPRISE CLUSTERS

Jyothi et al. 2016

Morpheus starts out with a really interesting premise - let's find out what users of big data clusters really care about, and then optimise for that! And it turns out that what users really care about is predictable job completion times. Morpheus monitors jobs that are periodically scheduled by a user, and _ infers_ a set of Service Level Objectives for them that a user can review and sign off on. From that point on, Morpheus takes over and manages the system so that jobs complete within their SLOs. The results are really impressive: an order of magnitude reduction in the number of jobs that miss their SLOs at the same cluster utilisation level, and with a ~20% reduction in the size of the cluster as well. Happier users, happier operators, and a happier CFO!

SIMPLE TESTING CAN PREVENT MOST CRITICAL FAILURES

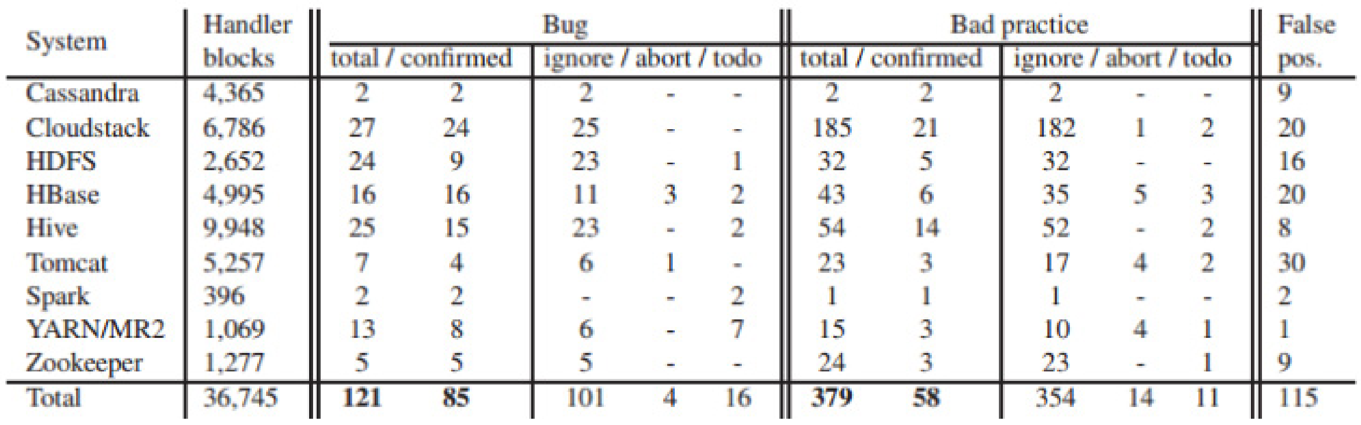
I first heard about "Simple testing can prevent most critical failures: an analysis of production failures in distributed data-intensive systems" by Yuan et al. (2014) through Caitie McCaffrey. It's a wonderful paper with information you can put to use straight away.

Yuan et al. 2014

The authors studied 198 randomly sampled user-reported failures from five data-intensive distributed systems (Cassandra, HBase, HDFS, MapReduce, and Redis). Each of these systems is designed to tolerate component failures and is widely used in production environments.

For each failure considered, we carefully studied the failure report, the discussion between users and developers, the logs and the code, and we manually reproduced 73 of the failures to better understand the specific manifestations that occurred. (All quotes from Yuan et al. 2014.)

There are a lot of really interesting findings and immediately applicable advice in this paper but the best of all for me is the discovery that the more catastrophic the failure (impacting all users, shutting down the whole system etc.), the simpler the root cause tends to be. In fact:



System	Handler blocks	Bug					Bad practice					False pos.
		total / confirmed		ignore / abort / todo			total / confirmed		ignore / abort / todo			
Cassandra	4,365	2	2	2	-	-	2	2	2	-	-	9
Cloudstack	6,786	27	24	25	-	-	185	21	182	1	2	20
HDFS	2,652	24	9	23	-	1	32	5	32	-	-	16
HBase	4,995	16	16	11	3	2	43	6	35	5	3	20
Hive	9,948	25	15	23	-	2	54	14	52	-	2	8
Tomcat	5,257	7	4	6	1	-	23	3	17	4	2	30
Spark	396	2	2	-	-	2	1	1	1	-	-	2
YARN/MR2	1,069	13	8	6	-	7	15	3	10	4	1	1
Zookeeper	1,277	5	5	5	-	-	24	3	23	-	1	9
Total	36,745	**121**	**85**	101	4	16	**379**	**58**	354	14	11	115

Table 1: Results of applying Aspirator to 9 distributed systems. If a case belongs to multiple categories (e.g. an empty handloer may also contain a "TODO" comment), we count it only once as an ignored exception. The "Handler blocks" column shows the number of exception handling blocks that Aspirator discovered and analyzed. "-" indicates Aspirator reported 0 warning.

Almost all catastrophic failures (92%) are the result of incorrect handling of non-fatal errors explicitly signalled in software.

And about a third of those (these are failures that brought down entire production systems, remember) are caused by trivial mistakes in error handling. They could have been detected by simple code inspection without even requiring any specific knowledge of the system.

Here are the three things you want to check for:

1) Error handlers that ignore errors (or just contain a log statement). This example led to data loss in HBase (Figure 1).

2) Error handlers with TODO or FIXME in the comment. This example took down a 4000-node production cluster (Figure 2).

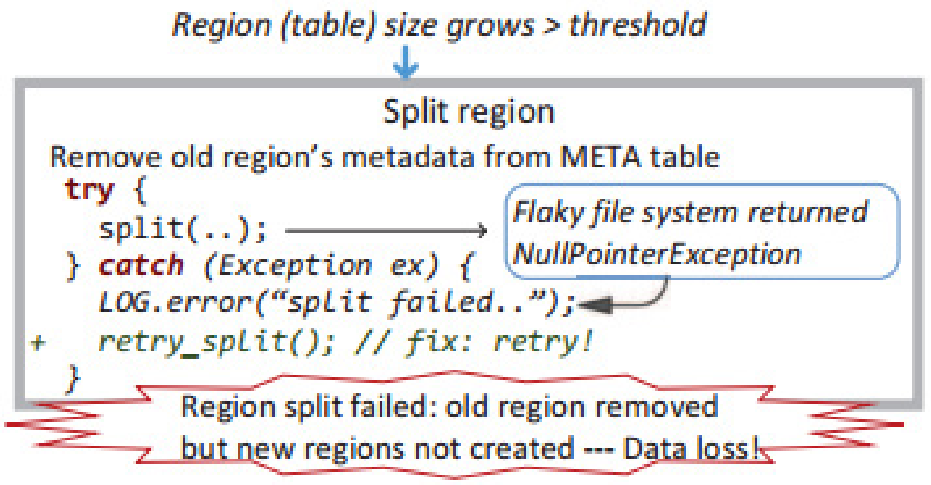
3) Error handlers that catch an abstract exception type (e.g. Exception or Throwable in Java) and then take drastic action such as aborting the system. This example brought down a whole HDFS cluster (Figure 3).

For these three scenarios, the authors built a static checking tool called Aspirator and used it to check some real-world systems, including those in the study.

If Aspirator had been used and the captured bugs fixed, 33% of the Cassandra, HBase, HDFS, and MapReduce's catastrophic failures we studied could have been prevented.

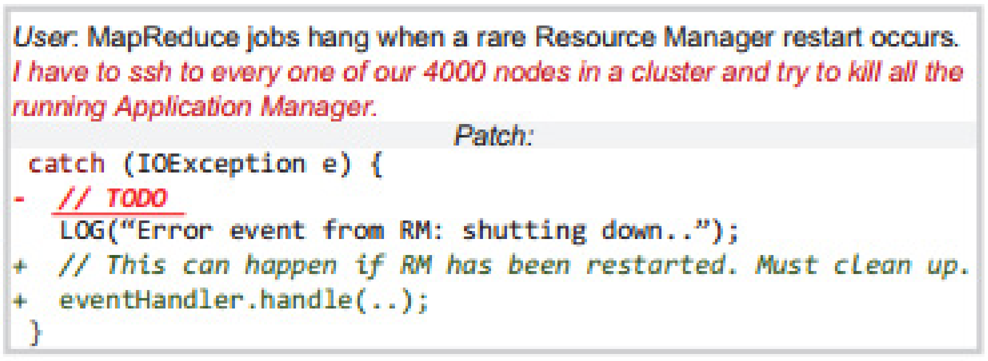
Using Aspirator to check the nine systems shown in table 1 found 500 new bugs and bad practices, along with 115 false positives. Aspirator reported 171 of these as bugs to developers. As of the publication of the paper, 143 had

Almost all catastrophic failures (92%) are the result of incorrect handling of non-fatal errors explicitly signalled in software.


Region (table) size grows > threshold
Split region
Remove old region's metadata from META table
Flaky file system returned NullPointerException
Region split failed: old region removed but new regions not created --- Data loss!

```
  try {
    split(..);
  } catch (Exception ex) {
    LOG.error("split failed..");
+   retry_split(); // fix: retry!
  }
```

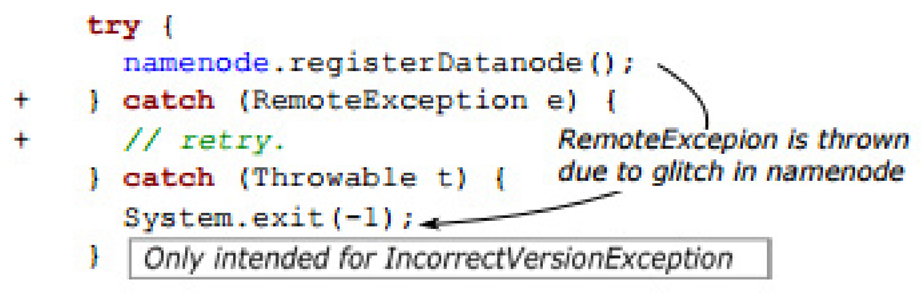
Figure 1: A data loss in HBase where the error handling was simply empty except for a logging statement. The fix was to retry in the exception handler.


User: MapReduce jobs hang when a rare Resource Manager restart occurs. *I have to ssh to every one of our 4000 nodes in a cluster and try to kill all the running Application Manager.*

Patch:

```
 catch (IOException e) {
-  // TODO
   LOG("Error event from RM: shutting down..");
+  // This can happen if RM has been restarted. Must clean up.
+  eventHandler.handle(..);
 }
```

Figure 2: A carastrophic failure in MapReduce where developers left a "TODO" in the error handler.


RemoteExcepion is thrown due to glitch in namenode
Only intended for IncorrectVersionException

```
    try {
      namenode.registerDatanode();
+   } catch (RemoteException e) {
+     // retry.
    } catch (Throwable t) {
      System.exit(-1);
    }
```

Figure 3: Entire HDFS cluster brought down by and over-catch.

been confirmed or fixed by the developers.

Another 57% of catastrophic failures caused by incorrect error handling do require some systems-specific knowledge to detect, but not much of such knowledge for a significant proportion of them.

In 23% of the catastrophic failures, while the mistakes in error handling were system specific, they are still easily detectable. More formally, the incorrect error handling in these cases would be exposed by 100% statement coverage testing on the error-handling logic. In other words, once the problematic basic block in the error-handling code is triggered, the failure is guaranteed to be exposed.

The final 34% of catastrophic failures do indeed involve complex bugs in error-handling logic. These are much harder to systematically root out but, hey — getting rid of the other two thirds of potentially catastrophe-inducing bugs is a pretty good start!

FAILURES ARE MOSTLY EASY TO REPRODUCE

Yuan et al. point out that a majority of the production failures (77%) can be reproduced by a unit test.

And if you do need a live system to reproduce a failure, you can almost certainly get away with a three-node system:

Almost all (98%) of the failures are guaranteed to manifest on no more than three nodes [and] 84% will manifest on no more than two nodes.... It is not necessary to have a large cluster of machines to expose bugs....

Furthermore, most failures require no more than three input events to get them to manifest: 74% of them are deterministic – that is, they are guaranteed to happen given the right input event sequences.

THE INFORMATION YOU NEED IS PROBABLY IN THE LOGS

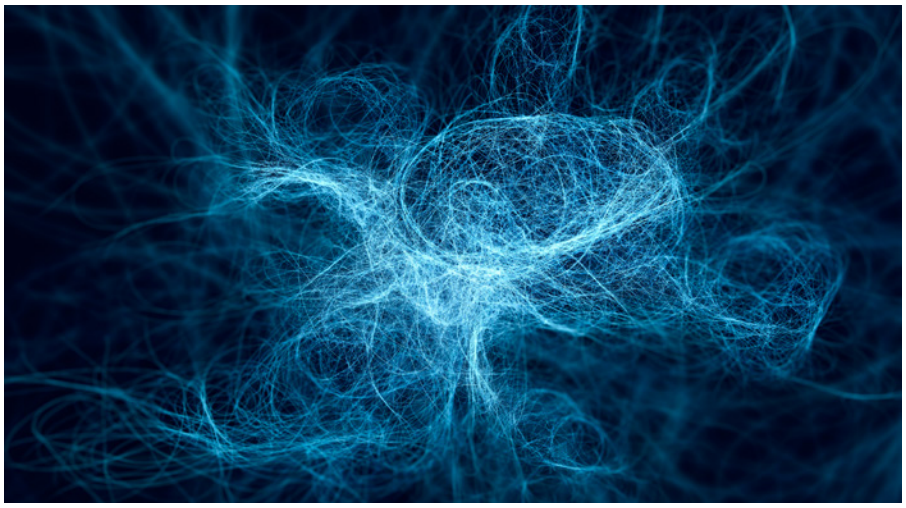
For a majority (84%) of the failures, all of their triggering events are logged. This suggests that it is possible to deterministically replay the majority of failures based on the existing log messages alone.

The logs are noisy, though: the median number of log messages printed by each failure was 824, and this is with a minimal configuration and a minimal workload sufficient to reproduce the failure.

TOWARDS DEEP SYMBOLIC REINFORCEMENT LEARNING

Every now and then I read a paper that makes a really strong connection with me, one where I can't stop thinking about the implications and which I can't wait to share with all of you. "Towards deep symbolic reinforcement learning" (Garnelo et al. 2016) is one such paper.

Garnelo et al. 2016



In the great seesaw of popularity for artificial intelligence techniques, symbolic reasoning and neural networks have taken turns, each having its dominant decade(s). The popular wisdom is that data-driven learning techniques (machine learning) won. Symbolic reasoning systems were just too hard and fragile to succeed at scale.

But what if we're throwing the baby out with the bath water? What if instead of having to choose between the two approaches, we could combine them: a system that can learn representations and then perform higher-order reasoning about those representations? Such combinations could potentially bring to bear the fullness of AI research over the last decades. I love this idea because:

1. It feels intuitively right (we as humans learn to recognise types of things, and then form probability-based rules about their behaviour for example).

2. It is very data efficient. To appropriate a phrase: "A rule can be worth 1000(+) data points!"

3. It opens up the possibility to incorporate decades of research into modern learning systems, where you can't help but think there would be some quick wins.

We show that the resulting system — though just a prototype — learns effectively, and, by acquiring a set of symbolic rules that are easily comprehensible to humans, dramatically outperforms a conventional, fully neural DRL system on a stochastic variant of the game. (All quotes from Garnelo et al. 2016.)

In short, this feels to me like something that could represent a real breakthrough and a step-change in the power of learning systems. You should take that with a pinch of salt because I'm just an interested outsider following along. I hope that I'm right though!

WHY COMBINE SYMBOLIC REASONING AND DEEP LEARNING?

Contemporary deep reinforcement learning (DRL) systems achieve impressive results but still suffer from a number of drawbacks:

- They inherit from deep learning the need for very large training sets, so that they learn very slowly.

- They are brittle in the sense that a trained network that performs well on one task often performs poorly on a new task, even if that new task is very similar to the original one.

- They do not use high-level processes such as planning, causal reasoning, or analogical reasoning to fully exploit the statistical regularities present in the training data.

- They are opaque — it is typically difficult to extract a human-comprehensible chain of reasons for the system's choice of action.

In contrast, classical AI uses language-like propositional representations to encode knowledge.

Thanks to their compositional structure, such representations are amenable to endless extension and recombination, an essential feature for the acquisition and deployment of high-level abstract concepts, which are key to general intelligence. Moreover, knowledge expressed in propositional form can be exploited by multiple high-level reasoning processes and has general-purpose application across multiple tasks and domains. Features such as these, derived from the benefits of human language, motivated several decades of research in symbolic AI. But as an approach to general intelligence, classical symbolic AI has been disappointing. A major obstacle here is the symbol-grounding problem.

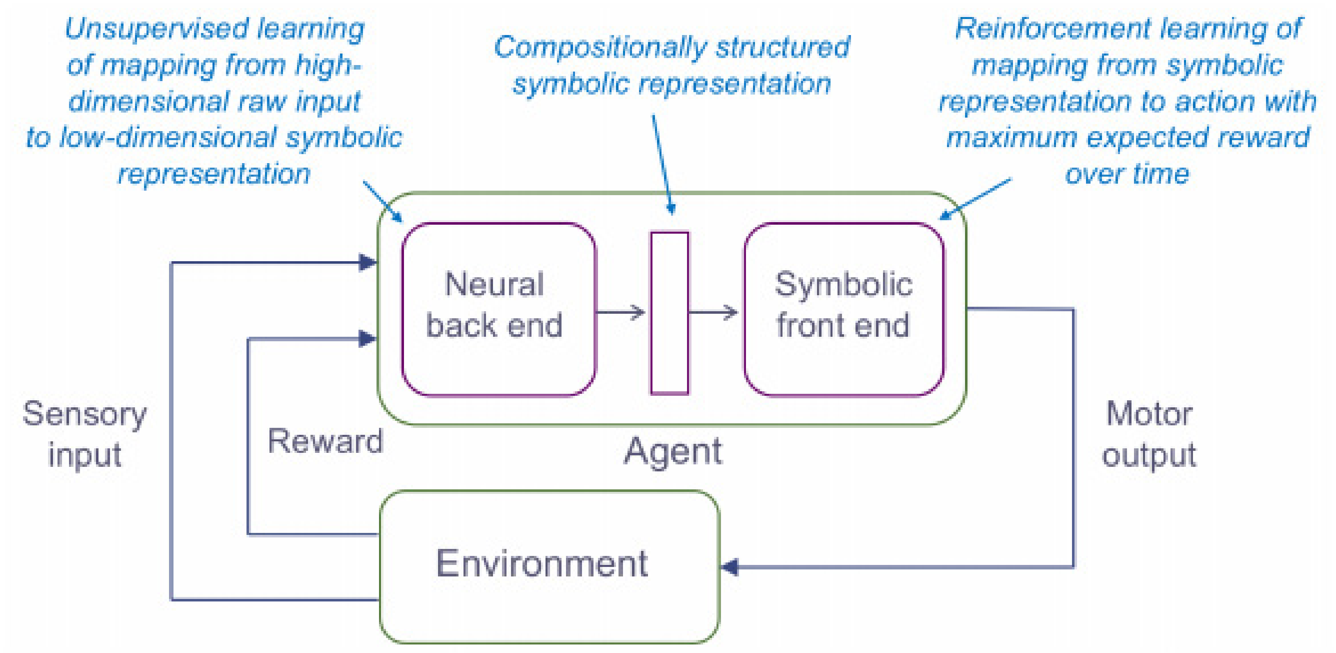
The symbol-grounding problem is this: where do the symbols come from? They are typically handcrafted rather than grounded in data from the real world. This brings a number of problems:

- They cannot support ongoing adaptation to a new environment.
- They cannot capture the rich statistics of real-world perceptual data.
- They create a barrier to full autonomy.

Machine learning has none of these problems!

So what if we could take the good bits from machine learning and combine them with the good bits from classical AI? Use machine learning to learn symbolic representations and then use symbolic reasoning on top of those learned symbols for action selection (in the case of DRL)?

You end with a system architecture that looks like this:


Unsupervised learning of mapping from high-dimensional raw input to low-dimensional symbolic representation
Compositionally structured symbolic representation
Reinforcement learning of mapping from symbolic representation to action with maximum expected reward over time
Neural back end
Symbolic front end
Sensory input
Reward
Agent
Motor output
Environment

PRINCIPLES FOR BUILDING DSRL SYSTEMS

There are four principles to follow when building deep symbolic reinforcement learning (DSRL) systems:

1. Support **conceptual abstraction** by mapping high-dimensional raw input into a lower-dimensional conceptual state space, then use symbolic methods that operate at a higher level of abstraction: "This facilitates both data efficient learning and transfer learning as well as providing a foundation for other high-level cognitive processes such as planning, innovative problem solving, and communication with other agents (including humans)."
2. Enable **compositional structure** that supports combining and recombining elements in an open-ended way: "To handle uncertainty, we propose probabilistic first-order logic for the semantic underpinnings of the low-dimensional conceptual state space rep-

What if we could take the good bits from machine learning and combine them with the good bits from classical AI? Use machine learning to learn symbolic rep-resentations and then use sym-bolic reasoning on top of those learned symbols for action selection (in the case of DRL)?

resentation onto which the neural front end must map the system's high-dimensional raw input.

3. Build on top of **common-sense priors.** It is unrealistic to expect an end-to-end reinforcement learning system to succeed with no prior assumptions about the domain. For example, objects frequently move, and typically do so in continuous trajectories: there are stereotypical events such as beginning to move, stopping, coming into contact with other objects, and so on.

4. Support **causal reasoning** through discovery of the causal structure of the domain and symbolic rules expressed in terms of both the domain and the common sense priors.

To carry out analogical inference at a more abstract level, and thereby facilitate the transfer of expertise from one domain to an-

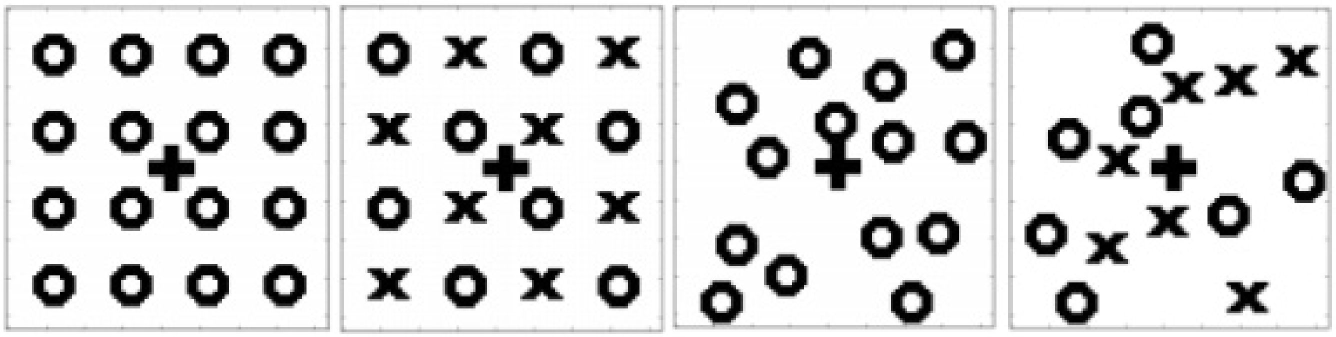


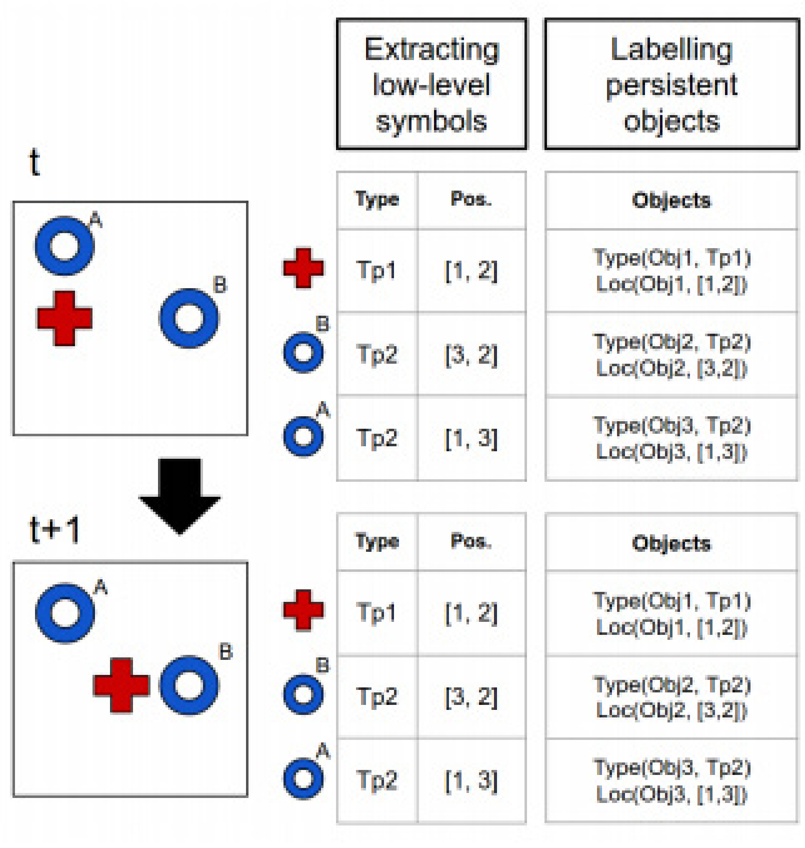
Figure 2: The four different game environments. The agent is represented by the '+' symbol. The static objects return positive or negative reward depending on their shape ('x' and 'o' respectively).

other, the narrative structure of the ongoing situation needs to be mapped to the causal structure of a set of previously encountered situations. As well as maximising the benefit of past experience, this enables high-level causal reasoning processes to be deployed in action selection, such as planning, lookahead, and off-line exploration (imagination).

A WORKED EXAMPLE

Well, it all sounds good, but does it actually work? The prototype system built by the authors learns to play four variations of a simple game. An agent moves around a square space populated by circles and crosses. It receives a positive reward for every cross it collects and a negative reward for every circle. The first variation has only circles in a fixed grid. The second has both circles and crosses in a fixed grid. The third has only circles in a random grid. The fourth has both circles and crosses in a random grid. (Figure 2)

The system has three stages: low-level symbol generation, representation building, and reinforcement learning.



t

Type	Pos.	Objects
Tp1	[1, 2]	Type(Obj1, Tp1) Loc(Obj1, [1,2])
Tp2	[3, 2]	Type(Obj2, Tp2) Loc(Obj2, [3,2])
Tp2	[1, 3]	Type(Obj3, Tp2) Loc(Obj3, [1,3])

t+1

Type	Pos.	Objects
Tp1	[1, 2]	Type(Obj1, Tp1) Loc(Obj1, [1,2])
Tp2	[3, 2]	Type(Obj2, Tp2) Loc(Obj2, [3,2])
Tp2	[1, 3]	Type(Obj3, Tp2) Loc(Obj3, [1,3])

Figure 3

The first stage use a convolutional neural network (CNN), a convolutional autoencoder, trained on 5,000 randomly generated sets of game objects scattered across the screen. The activations in the middle layer of the CNN are used directly for detection of objects in the scene. The salient areas in an image result in higher activation throughout the layers of the CNN. Given the geometric simplicity of the games, this is enough to extract the individual objects from any given frame.

The objects identified this way are then assigned a symbolic type according to the geometric properties computed by the autoencoder. This is done by comparing the activation spectra of the salient pixels across features.

At this point, the CNN knows the type and position of the objects. (Figure 3)

The second stage learns to track objects across frames in order to learn from their dynamics. The system is supported in this task by a common-sense prior: object persistence over time. This is broken down into three measures and ultimately combined into a single value: how close an object in one frame is to an object in another frame (spatial proximity is an indicator it may be the same object); how likely it is that an object transformed from one type into another (by learning a transition probability matrix); and what change there is in the neighbourhood of an object.

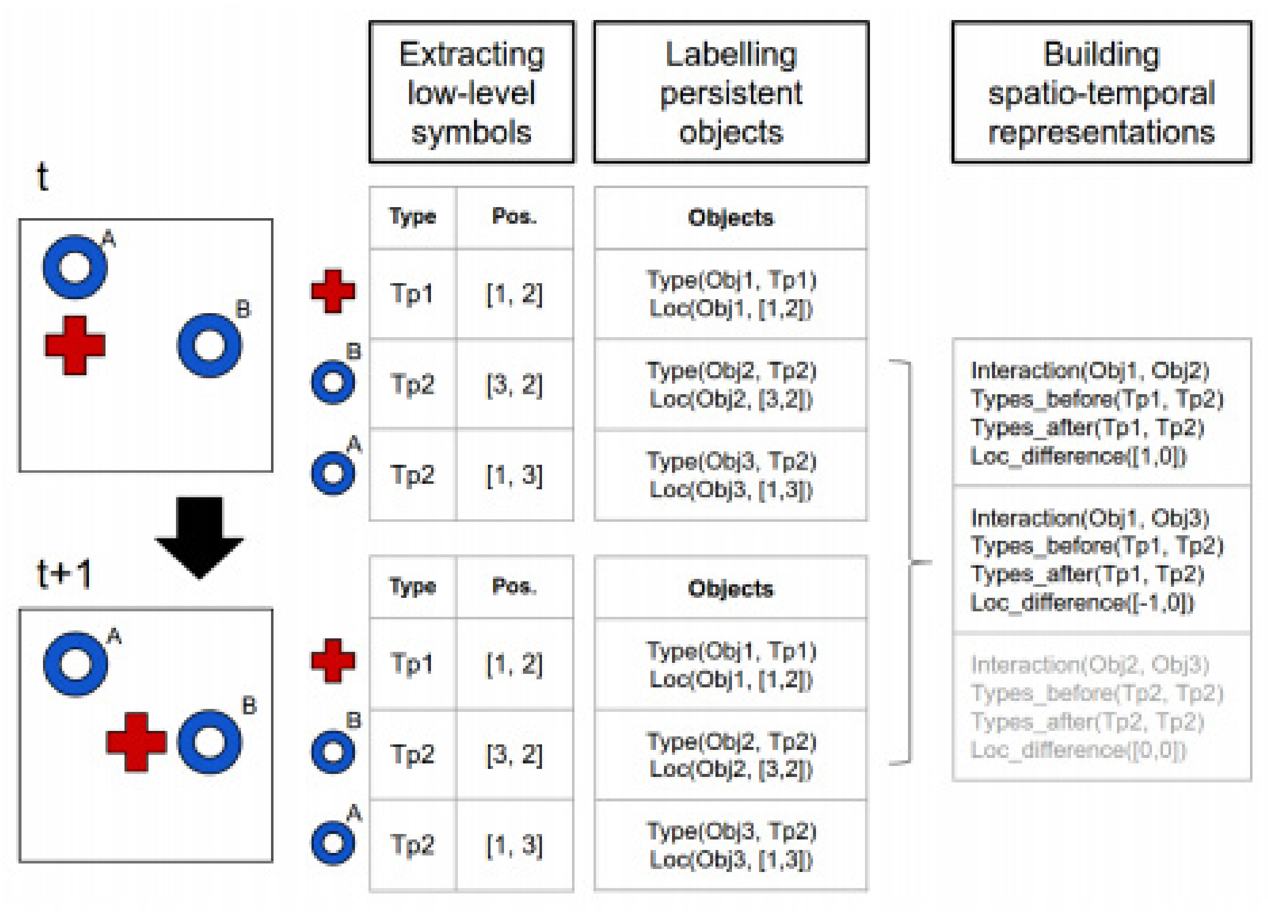


Figure 4

The representation learned at the end of the second stage differs from that of the first stage in two key ways:

- It is extended to understand changes over time.
- Positions of objects are represented by relative coordinates to other objects rather than by absolute coordinates: “This approach is justified by the common-sense prior that local relations between multiple objects are more relevant than the global properties of single objects.”

We now have a concise spatio-temporal representation of the game situation, one that captures not only what objects are in the scene along with their locations and types, but also what they are doing. In particular, it represents frame-to-frame interactions between objects, and the changes in type and relative position that result. This is the input to reinforcement learning, the third and final stage of the pipeline. (Figure 4)

Finally, the CNN enters the reinforcement learning stage, where the relative location representation of objects greatly reduces the state space, on the assumption that things that are far apart (both in the game and in the real world) tend to have little influence on each other.

In order to implement this independence, we train a separate Q function for each interaction between two object types. The main idea is to learn several Q functions for the different interactions and query those that are relevant for the current situation. Given the simplicity of the game and the reduced state space that results from the sparse symbolic representation, we can approximate the optimal policy using tabular Q-learning.

EVALUATION RESULTS

You can use precision and recall metrics to evaluate agent performance, where precision is interpreted as the percentage of collected objects that are positive (crosses) and recall is interpreted as the percentage of available positive objects that have been collected.

The first results show that the DSRL agent improves with training to a precision of 70%. It's interesting to compare the performance of DQN (a conventional DRL system) and the DSRL system. DQN performs almost perfectly in the grid scenario, but struggles when objects are position at random to learn an effective policy within 1,000 epochs. The DSRL system performs much better on this game variant:

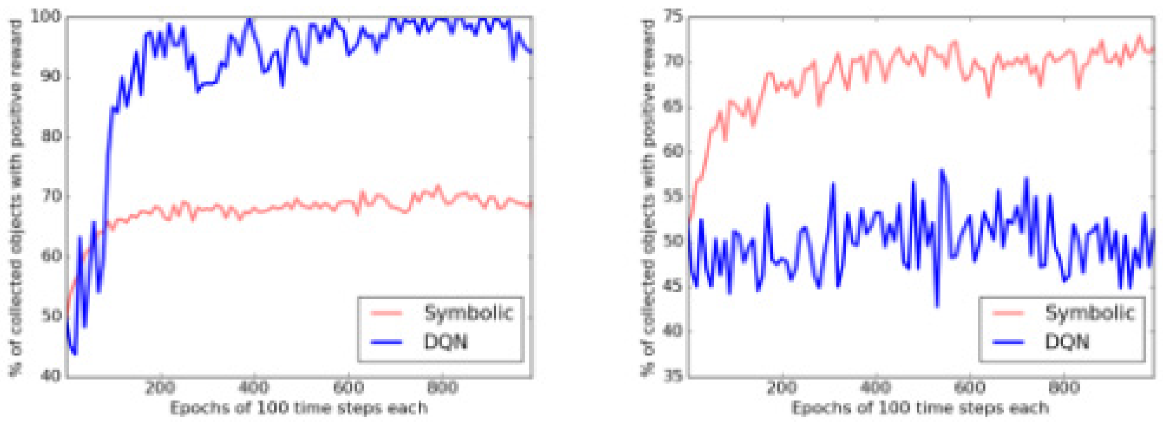


Figure 6: Comparison between DQN and symbolic approach. Average percentage of objects collected over 200 games that return positive reward in the grid environment (left) and in the random environment (right).

To evaluate transfer learning, both DQN and DSRL were trained on the grid variation and then tested on the random variant. DQN does no better than chance, whereas the DSRL system is able to approach 70% precision.

We conjecture that DQN struggles with this game because it has to form a statistical picture of all possible object placements, which would require a much larger number of games. In contrast, thanks to the conceptual abstraction made possible by its symbolic front end, our system very quickly "gets" the game and forms a set of general rules that covers every possible initial configuration. This demonstration merely hints at the potential for a symbolic front end to promote data-efficient learning, potential that we aim to exploit more fully in future work. Our proof-of-concept system also illustrates one aspect of the architecture's inherent capacity for transfer learning.

Moreover, the DSRL system can explain its actions: every action choice can be analysed in terms of the Q functions involved in the decision. These Q functions describe the types of objects involved in the interaction as well as their relations, so we can track back to the reasons that led to a certain decision.

AND THIS IS JUST THE BEGINNING...

We could wire in a more sophisticated deep network capable of unsupervised learning to disentangle representations in the earlier stages.

We can exploit many more of the achievements in classical AI, notably the incorporation of inductive-logic programming, formal techniques for analogical reasoning, and building in a planning component that exploits the knowledge of the cause structure of the domain acquired during the learning process.

In domains with sparse reward, it's often possible for an agent to discover a sequence of actions leading to a reward state through off-line search rather than on-line exploration. Contemporary logic-based planning methods are capable of efficiently finding large plans in complex domains, and it would be rash not to exploit the potential of these techniques.

Finally, the symbolic components of the proposed architecture one day could well be using neural-based implementations of symbolic reasoning functions.

In the mean time, an architecture that combines deep neural networks with directly implemented symbolic reasoning seems like a promising research direction.

CSI MEETS PUBLIC WI-FI: INFERRING YOUR SMARTPHONE PASSWORD VIA WI-FI SIGNALS

Not that CSI. CSI in "When CSI meets public WiFi: Inferring your mobile phone password via WiFi signals" (Li et al. 2016) stands for "channel state information", which represents the state of a wireless channel in a signal transmission process.

Li et al. 2016

WindTalker is motivated from the observation that keystrokes on mobile devices will lead to different hand coverage and the finger motions, which will introduce a unique interference to the multi-path signals and can be reflected by the channel state information (CSI). ***(All quotes from Li et al. 2016.)***

By setting up a rogue access point, determining when a user is entering a PIN (for the Alipay payment system, the largest mobile-payments company in the world, in the demonstrated attack), and observing the fluctuations in Wi-Fi signal, it's possible to recover the PIN entered.

Particularly with side-channel attacks, I usually feel a mix of "Wow, you can do that! That's really ingenious!" coupled with a sense of despair at just how insecure everything really is in the presence of skilled attackers. This paper is no exception.

WindTalker is the latest in a long line of keystroke/PIN inference methods. Owusu et al. demonstrated accelerometer-based keystroke inference that recovers six-character passwords on smartphones. Liu et al. applied a similar idea to use a smart watch to track hand movements over a keyboard and achieve 65% recognition accuracy. Zhu et al. showed that smartphone microphones can be used to record the unique acoustic sound of keystrokes. Liu et al. exploited smartphone audio hardware to recover 94% of keystrokes. Yue et al. demonstrated Google Glass/webcam-based keystroke inference with a success rate over 90%. Shukla et al.'s video-based attack breaks over 50% of PINs. WiKey uses CSI waveform patterns to distinguish keystrokes on an external keyboard. WiPass detects graphical unlock passwords. (See section 8.2 in Li et al. for these references.) But WindTalker is particularly effective because it doesn't require any access to the victim's phone, it works with regular mobile phones, and it piggybacks on an existing Wi-Fi connection.

The basic outline of the attack is as follows:

- Set up a public Wi-Fi hotspot where a target is likely to remain relatively stationary and within ~1.5 m of the hotspot. Target a particular table in café, for example.

- When the target connects, monitor their Wi-Fi traffic. This is (hopefully!) transmitted over HTTPS, but the meta data is not encrypted, and that's enough for WindTalker to work with.

WindTalker needs to determine the point in time at which the target is about to enter a PIN — for example, when completing an Alipay mobile payment. It turns out that an IP address is sufficient information.

To determine the sensitive input windows, WindTalker runs in a real-time fashion to collect the meta data (e.g., IP address) of the targeted sensitive mobile payment applications (e.g., Alipay). For example, in the experiment, Alipay applications will always route their data to the server of some specific IP address such as "110.75.xx.xx". This IP address will be kept to be relatively stable for one or two weeks. With the traffic meta data, WindTalker obtains the rough start time and end point of the sensitive input window via searching packets whose destination is "110.75.xx.xx". Then WindTalker begins to analyze the corresponding CSI data in that period of time.

- During the sensitive input window, WindTalker sends ICMP Echo Requests to the target's smartphone (about 800 packets a second). The phone replies with an Echo Reply. For a 98-byte ICMP packet sent at 800 packets/second, the bandwidth consumption is only 78.4 kB/s, which will not noticeably degrade the Wi-Fi experience of the target.

- At the end of the input window, analyse the data, and recover the PIN!

I guess that last part warrants a little further explanation...

First, we need to clean up some of the noise in the data. This is achieved using a low-pass filter to remove high-frequency noise, since the variations in CSI waveforms caused by finger motion lie at the low end of the spectrum. The next step is dimension reduction using principle-component analysis (PCA). PCA is a well-established algorithm for finding the most significant/influential components in the CSI time series.

With PCA, we can identify the most representative components influenced by the victim's hand and fingers movement and remove the noisy components at the same time. In our experiment, it is observed the first k = 4 components show the most significant changes in CSI streams and the rest of the components are noise.

Now that we have a cleaned up CSI waveform, it's time to start identifying the individual finger touches within it. This is done with a three-step process: (Figure 1)

The starting point is shown in (a) above. Another filter (a Butterworth filter) is applied with a 10-Hz cut-off to smooth the waveform (b). The CSI waveforms for individual keystrokes now need to be extracted (they're easy to spot by eye, indicating that this should not be too difficult!):

It is observed that the CSI segments during the keystroke peri-

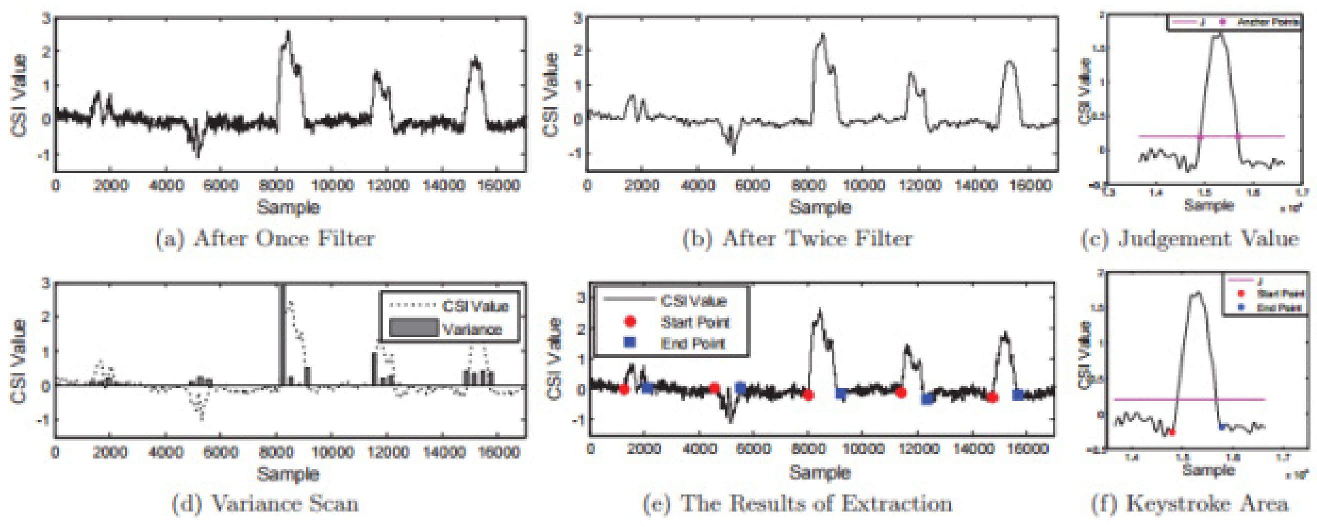


Figure 1: Keystroke Extraction

od show a much larger variance than those happening out of the period, which is shown in Fig.1(d).

So we extract segments where variance is greater than the pre-determined threshold shown in (c).

Given a keystroke segment, the final stage is to work out what keystroke it represents.

As shown in Fig.2, it is observed that different keystrokes will lead to different waveforms, which motivates us to choose waveform shape as the feature for keystroke classification. To compare the waveforms of different keystrokes, we adopt dynamic time warping (DTW) to measure the similarity between the CSI time series of two keystrokes. However, directly using the keystroke waveforms as the classification features leads to high computational costs in the classification process since waveforms contain many data points for each keystroke. Therefore, we leverage discrete wavelet transformation (DWT) to compress the length of [the] CSI waveform....

Finally, a classifier uses the DTW distances between the input waveform and the key-number waveforms in a reference data-set. The classifier chooses the key number with the minimum score as the predicted key number, but all scores are saved in order to generate password candidates.

Where do those reference key number waveforms come from?

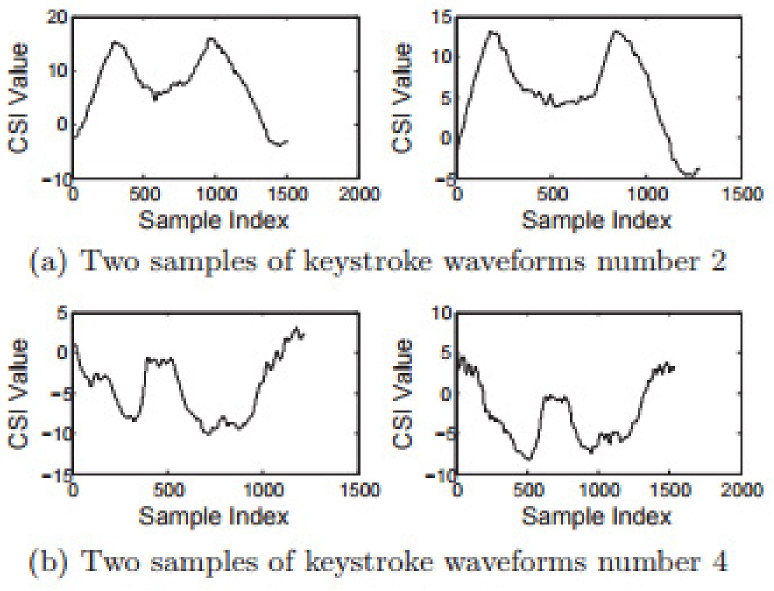
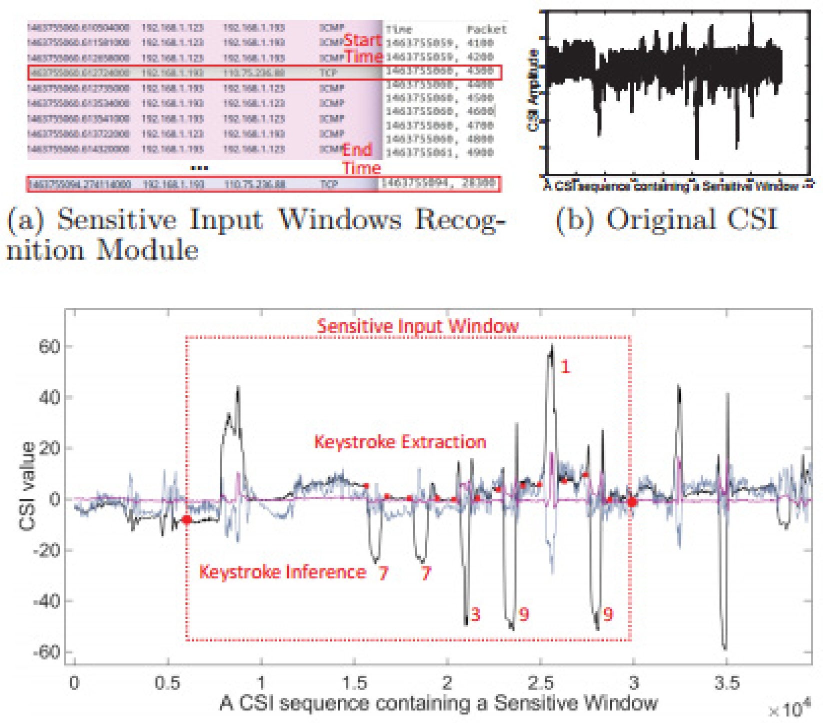


Figure 2: CSI Difference Between Two Number

Table 1: Recovery Rate and Loop Times

Loop Times	One	Three	Five	Ten
Recovery Rate	68.3%	73.3%	78.3%	81.7%



(a) Sensitive Input Windows Recognition Module (b) Original CSI

(c) Keystroke Inference Module

Figure 3: WindTalker in Case Study

The way your hand moves interferes with the wifi signal of your phone and... oops there goes your PIN!

Right now, WindTalker needs a per-user pre-training session. With enough data from enough users, perhaps that won't be required in the future. It seems a bit much to ask a target to pre-train on your system so that you can nab their password! But the pre-training required is very minimal, with just one sample of a given keystroke, the recovery rate is as high as 68.3%.

In practice, the attackers have more choices to achieve the user-specific training. For example, they can simply offer the user free Wi-Fi access and, as the return, the victim should finish the online training by clicking the designated numbers. It can also mimic a text captcha to require the victim to input the chosen numbers. We further analyze the impact of the number of training data on recovery rate in WindTalker. Table 1 shows the recovery rate increases with the training loop increases. Even if there is only one training sample for a keystroke, WindTalker can still achieve whole recovery rate of 68.3%.

The paper concludes with a case study of cracking Alipay PINs with volunteer users. The system thus knows to look for segments inside a sensitive input window with six keystrokes (Figure 3).

After the keystroke extraction and recognition processs, WindTalker lists possible password candidates. In the example above, the top three candidates suggested by WindTalker were 773919, 773619, and 773916. The actual password was indeed 773919. Especially if an application allows multiple attempts at entering a passcode, there's a good chance that WindTalker will get the right one.

KRAKEN

USING LIVE TRAFFIC TESTS TO RESOLVE RESOURCE BOTTLENECKS IN LARGE-SCALE WEB SERVICES

How do you know how well your systems can perform under stress? How can you identify resource-utilisation bottlenecks? And how do you know your tests match the conditions of live production traffic? You could try modelling load (simulation) but it's not really feasible to model systems undergoing constantly evolving workloads, frequent software release cycles, and complex dependencies.

Veeraraghavan et al. 2016

That leaves us with load testing. When load testing, we have two further challenges:

- We must ensure that the load-test workload is representative of real traffic. This can be addressed using shadow traffic: replaying logs in a test environment.
- We must deal with side effects of the load testing that may propagate deep into the system. This can be addressed by stubbing, but it's hard to keep up with constantly changing codebases, and reduces the fidelity of end-to-end tests by not stressing dependencies that otherwise would have been affected.

Facebook use a very Chaos Monkey-esque approach to this problem, described in "Kraken: Leveraging live traffic tests to identify and resolve resource utilization bottlenecks in large scale web services" (Veeraraghavan et al. 2016). The company simply runs load tests using live traffic with a system called Kraken, and has been doing so for about three years. The core idea is really simple — just update the routing layer to direct more traffic at the systems you want to test. Here are four things to like about the approach:

- Live traffic does a *very* good job of simulating the behaviour of live traffic.
- You don't need any special test setups.
- Live-traffic tests expose bottlenecks that arise due to complex system dependencies.
- It forces teams to harden their systems to handle traffic bursts, overloads etc., thus increasing the system's resilience to faults.

On that last point, it's easy to see how a busy service team might ignore (deprioritise) a performance report from a specialist performance team stressing their system. It's a whole different thing when you *know* your system is going to be pushed to its limits in production at some point. It's just like when you know a chaos monkey is likely to crash a component at some point.

And of course, just as with Chaos Monkey, it's an idea that can sound scary as hell when you first hear about it. It's why the paper stresses, "Safety is a key constraint when working with live traffic on a production system." (All quotes from Veeraraghavan et al. 2016.)

WHY IS CONTINUOUS LOAD TESTING IMPORTANT?

Frequent load testing of services is a best practice that few follow today, but that can bring great benefits.

The workload of a web service is constantly changing as its user base grows and new products are launched. Further, individual software systems might be updated several times a day or even continually.... An evolving workload can quickly render models obsolete.

Not only that, but the infrastructure systems supporting a given service constantly change too, and a data centre may be running hundreds of software systems with complex interactions.

Running frequent performance tests on live systems drove a number of good behaviours and outcomes at Facebook:

It highlighted areas with non-linear responses to traffic increases, and where there was insufficient information to diagnose performance problems.

It encouraged subsystem developers to identify system-specific counters for performance, error rate, and latency that could be monitored during a test:

We focused on these three metrics (performance, error rates, and latency) because they represent the contracts that clients of a service rely on — we have found that nearly every production system wishes to maintain or decrease its latency and error rate while maintaining or improving performance.

It changed the culture (with deliberate effort), shifting from load tests as a painful event for a system to survive to something that developers looked forward to as an opportunity to better understand their systems.

It improved monitoring and the understanding of which metrics are the most critical bellwethers of non-linear behaviours.

Performance testing as a regular discipline dramatically improved

the capacity of Facebook's systems over time:

Kraken has allowed us to identify and remediate regressions, and address load imbalance and resource exhaustion across Facebook's fleet. Our initial tests stopped at about 70% of theoretical capacity, but now routinely exceed 90%, providing a 20% increase in request serving capacity.

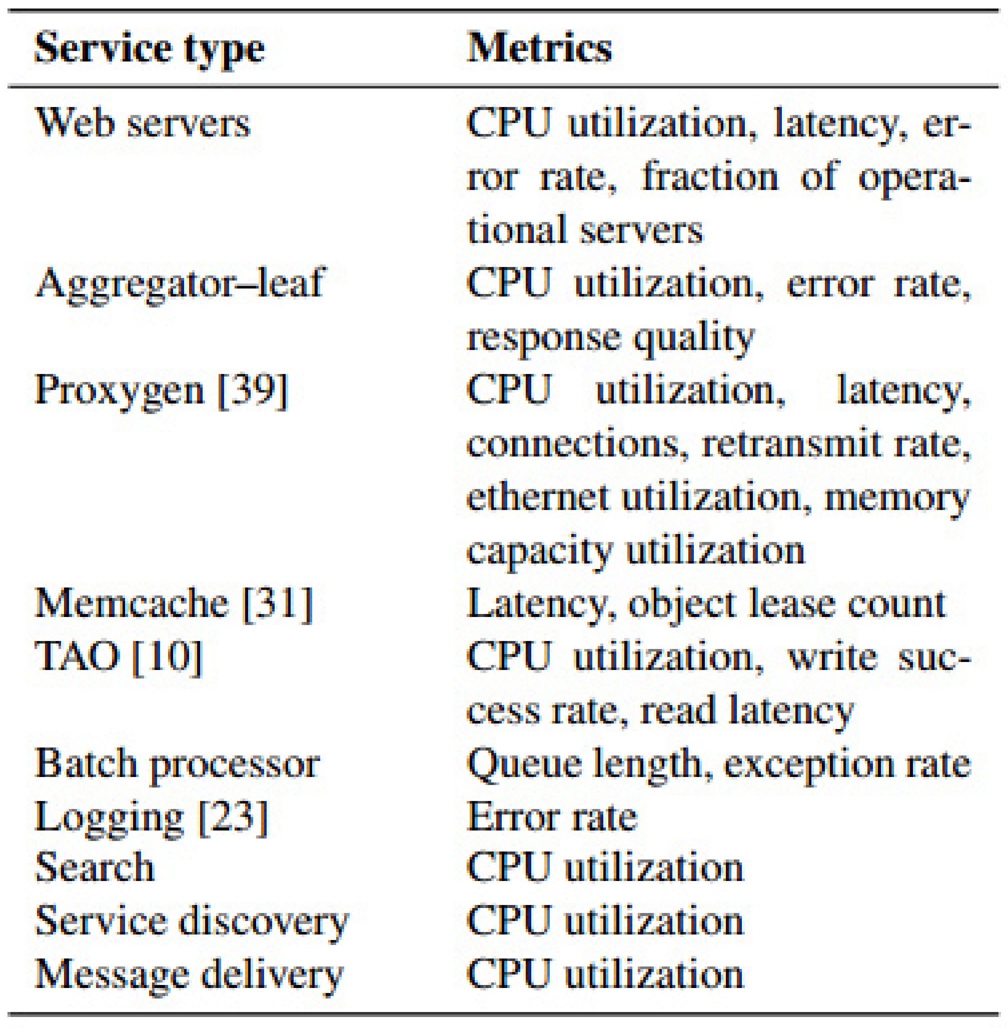
A 20% increase is a big deal when you think of all the costs involved in operating systems at Facebook scale!

Getting to that 20% increase does of course require work to address the findings resulting from the load tests. As a short aside, one of the companies I'm involved with, Skipjaq, is also a big believer in the benefits of frequent performance testing. It looks at individual services and automates the exploration of the configuration space for those services (OS, JVM, server process settings, etc.) to optimise performance on or across infrastructure with no code changes required. Performance tuning at this level is something of a black art, which most teams are either not equipped or don't have time to deal with, given the constantly changing workloads and codebases as previously described. Early results with real customer applications and workloads suggest there are big performance gains to be had here for many applications too.

HOW DO YOU LOAD-TEST SAFELY USING LIVE PRODUCTION SYSTEMS?

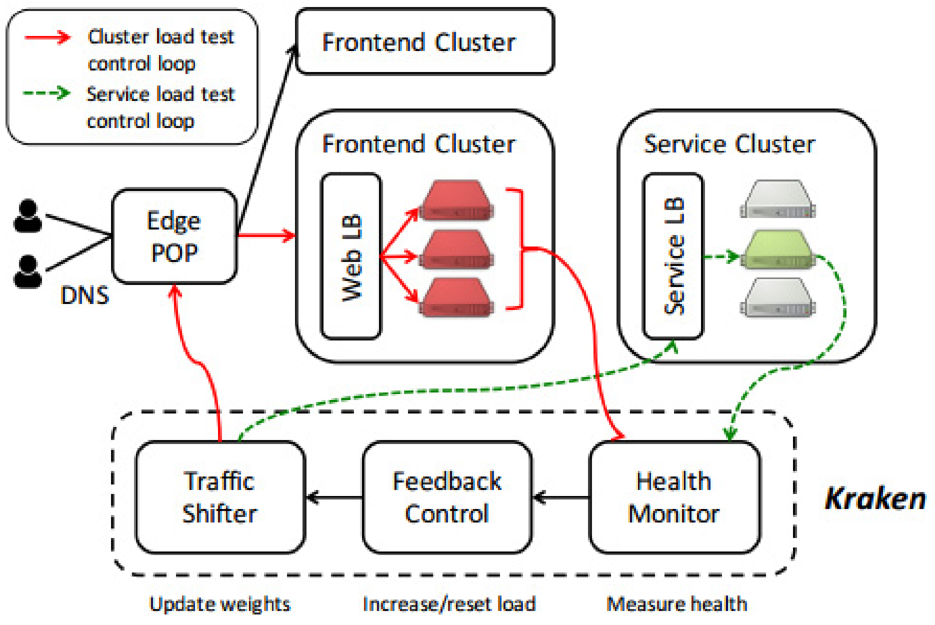
Load testing with live production systems (and traffic, so every request matters) requires careful monitoring and live traffic adjustment.

Our insight was that Kraken running on a data center is equivalent to an operational issue affecting the site — in both cases our goal is to provide a good user experience. We use two metrics, the web servers' 99th-percentile response time and HTTP fatal-error rate, as proxies for the user experience, and determined in most cases this was adequate to avoid bad outcomes. Over time, we have added other metrics to improve safety such as the median queueing delay on web servers, the 99th-percentile CPU utilization on cache machines, etc. Each metric has an explicit threshold demarcating the vitality of the system's health. Kraken stops the test when any metric reaches its limit, before the system becomes unhealthy.



Service type	Metrics
Web servers	CPU utilization, latency, error rate, fraction of operational servers
Aggregator–leaf	CPU utilization, error rate, response quality
Proxygen [39]	CPU utilization, latency, connections, retransmit rate, ethernet utilization, memory capacity utilization
Memcache [31]	Latency, object lease count
TAO [10]	CPU utilization, write success rate, read latency
Batch processor	Queue length, exception rate
Logging [23]	Error rate
Search	CPU utilization
Service discovery	CPU utilization
Message delivery	CPU utilization

Table 1: Health metrics for various systems that are affected by web load.



THE ESSENCE OF KRAKEN

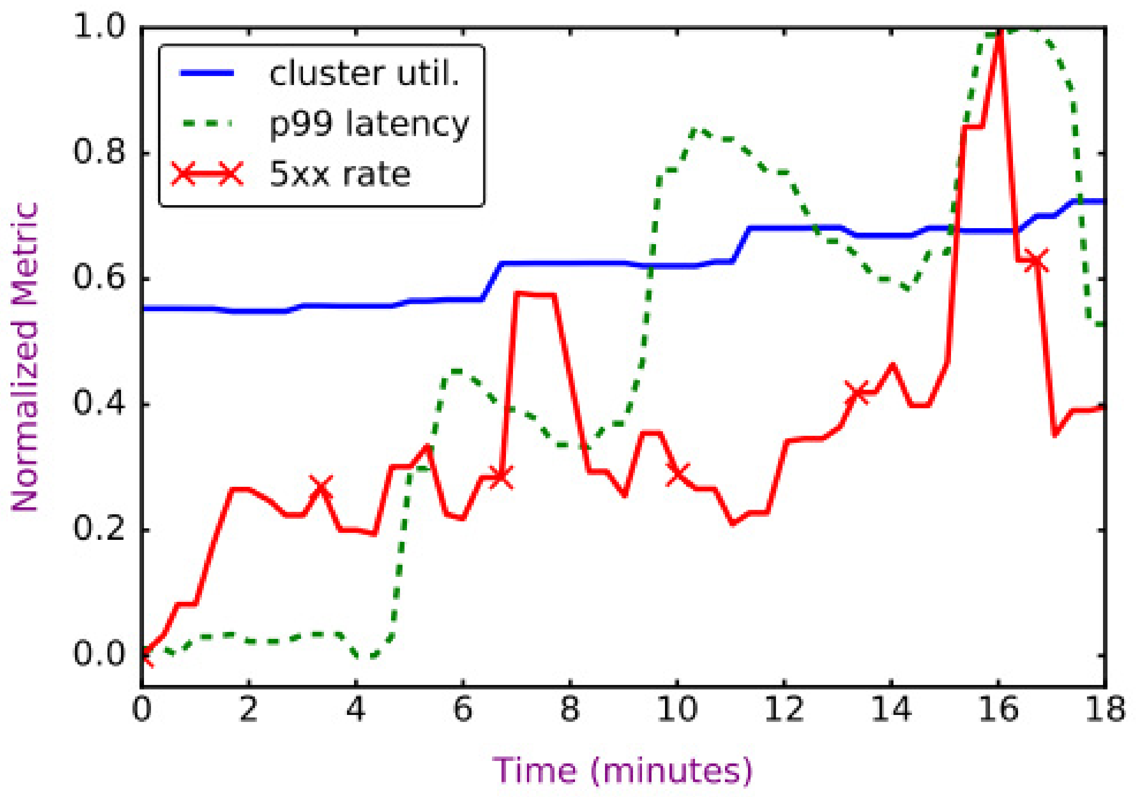
Kraken shifts traffic in the Facebook routing infrastructure by adjusting the weights that control load balancing. The edge weights control how traffic is routed from a POP to a region, the cluster weights control routing to clusters within regions, and server weights balance across servers within a cluster. Kraken sits as part of a carefully designed feedback loop (red lines in the figure below) that evaluates the capacity and behaviour of the system under test to adjust the stress it is putting on systems. The traffic-shifting module queries Gorilla for system health before determining the next traffic shift to the system under test. Health-metric definitions themselves are stored in a distributed configuration management system.

*At the start of a test, Kraken aggressively increases load and maintains the step size while the system is healthy. We have observed a trade-off between **the rate of load increase** and **system health**. For systems that employ caching, rapid shifts in load can lead to large cache miss rates and lower system health than slow increases in load. In practice, we find that initial load-increase increments of around 15% strike a good balance between load-test speed and system health.*

As system health metrics approach their thresholds, the load increases are dramatically reduced, down to increments of 1%. This allows the capture of more precise capacity information at high load.

Here's a sample cluster load test. Every five minutes, Kraken inspects cluster health and decides how to shift traffic. It takes about two minutes for a load shift to occur and the results to be seen. The test stopped when the p99 latency (red line) exceeded its threshold level for too long.

Kraken performs load testing with live production traffic by adjusting the weights that control load balancing, concentrating traffic in the areas to be tested.



In the example above, the system only hit a peak utilization of 75%, well below Facebook's target of 93%. The charts below show how Kraken can start to explain performance gaps. Chart (a) shows the widening gap between cluster load-test capacity as measured and the theoretical capacity, and (b) shows the increasing time spent waiting for cache responses as cluster load increases.

Section 5 in the paper is packed full of examples of issues found by Kraken and how they were resolved.

Lessons learned

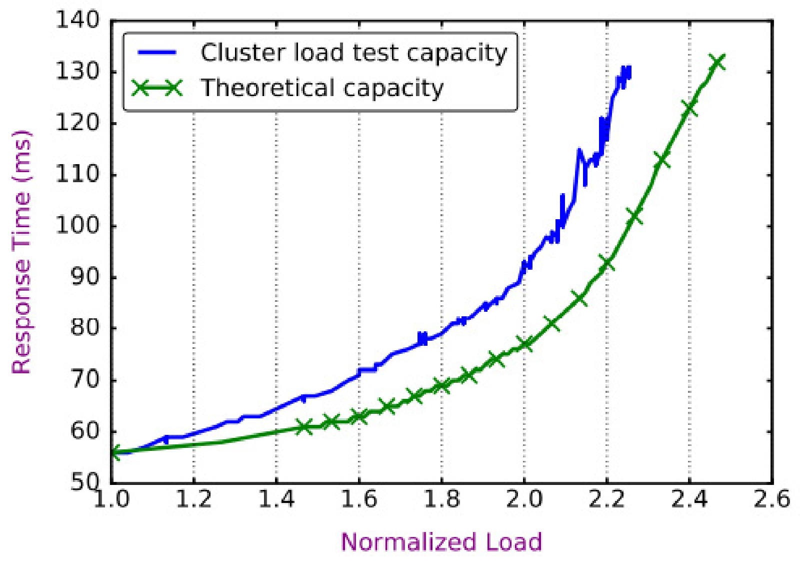
We have learned that running live-traffic load tests without compromising on system health is difficult. Succeeding at this approach has required us to invest heavily in instrumenting our software systems, using and building new debugging tools, and encouraging engineers to collaborate on investigating and resolving issues.

Simplicity is key to Kraken's success — the stability of simple systems is needed to debug complex issues.

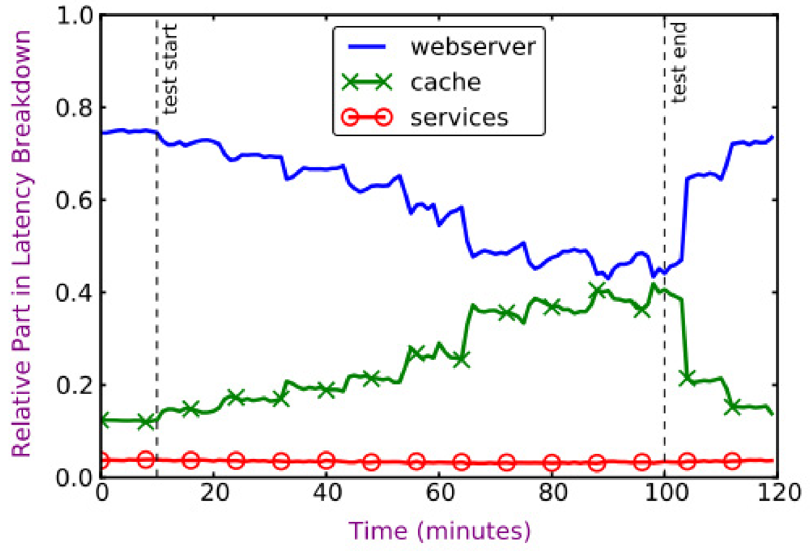
It is difficult to identify the right metrics to capture a system's performance, error rate, and latency:

We have found it useful to identify several candidate metrics and then observe their behaviour over tens to hundreds of tests to determine which ones provide the highest signal. However, once we identify stable metrics, their thresholds are easy to configure and almost never change once set.

Specialized error-handling mechanisms such as automatic failover and fall backs make systems harder to debug:



(a) Performance gap between cluster load test capacity and theoretical capacity.



(b) Latency breakdown in cluster load test.

We find that such mitigations need to be well instrumented to be effective in the long run, and prefer more direct methods such as graceful degradation.

There are quick fixes (allocating capacity, changing configuration, or load-balancing strategies) that have been essential for rapidly solving production issues. Profiling, performance tuning, and system redesign are only undertaken when the benefit justifies the cost.

I'll leave you with this quote:

…Kraken allowed us to surface many bottlenecks that were hidden until the systems were under load. We identified problems, experimented with remedies, and iterated on our solutions over successive tests. Further, this process of continually testing and fixing allowed us to develop a library of solutions and verify health without permitting regressions.

MORPHEUS: TOWARDS AUTOMATED SLOS FOR ENTERPRISE CLUSTERS

I'm really impressed with "Morpheus: Towards automated SLOs for enterprise clusters" (Jyothi et al. 2016) — it covers all the bases from user studies to find out what's really important to end users to data-driven engineering, a sprinkling of algorithms, a pragmatic implementation being made available in open source, and of course, impactful results. If I were running a big-data cluster, I'd definitely be paying close attention!

Jyothi et al. 2016

Morpheus is a cluster scheduler that optimises for user satisfaction (we don't see that often enough as a goal!) and along the way reduces the number of jobs that miss their deadlines by up to an order of magnitude while maintaining cluster utilisation and lowering cluster footprint by 14-28%.

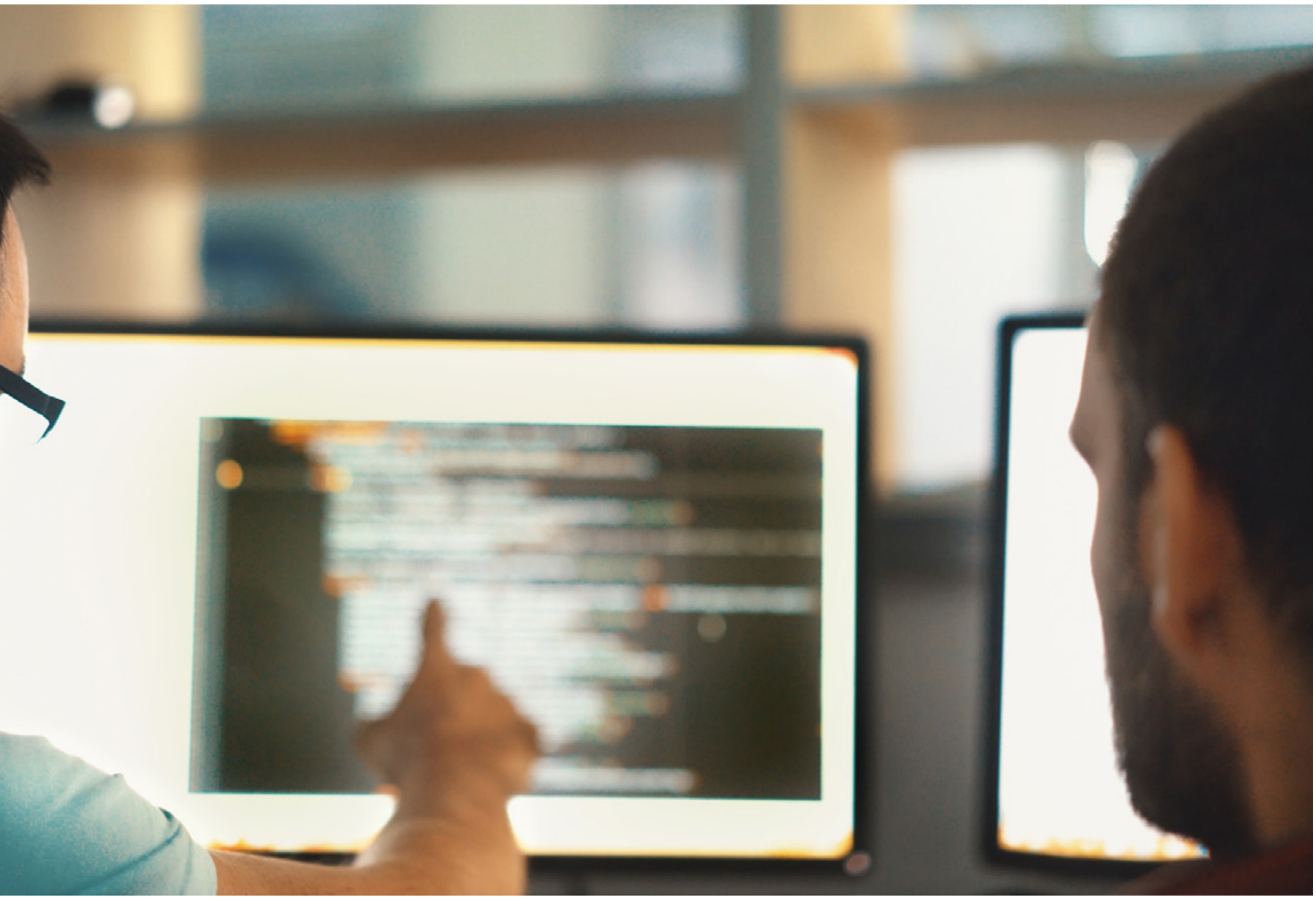
In this paper, we present Morpheus, a system designed to resolve the tension between predictability and utilization — that we discovered thorough analysis of cluster workloads and operator/user dynamics. Morpheus builds on three key ideas: automatically deriving SLOs and job resource models from historical data, relying on recurrent reservations and packing algorithms to enforce SLOs, and dynamic reprovisioning to mitigate inherent execution variance. ***(All quotes from Jyothi et al. 2016.)***

WHAT DO USERS OF BIG-DATA CLUSTERS CARE ABOUT?

The team gathered some interesting insights into what goes on in (Microsoft's?) big-data clusters by:

- analysing execution logs from millions of jobs running on clusters with over 50K nodes;

User of big data clusters care about predictable job completion by a deadline.



- looking at infrastructure deployment and upgrade logs;
- end-user interviews and analysis of discussion threads and escalation tickets from end-users, operators, and decision makers; and
- a selection of targeted micro-benchmarks.

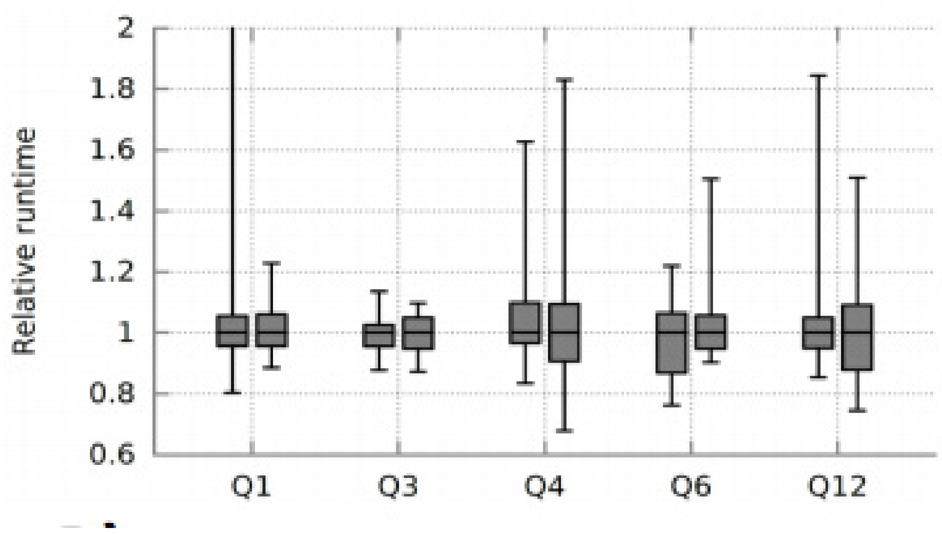
The findings are fascinating.

Based on interviews with cluster operators and users, we isolate one observable metric which users care about: job completion by a deadline. Specifically, analysing the escalation tickets, some users seem to form expectations such as: "95% of job X runs should complete by 5 pm". Other users are not able to specify a concrete deadline, but do state that other teams rely on the output of their job, and may need it in a timely manner.

- Over 75% of the workload is production jobs.
- Users are 120 times more likely to complain about performance (un)predictability than about fairness.
- Over 60% of the jobs in the largest clusters are recurrent. Most of these recurring jobs are production jobs operating on continuously arriving data, hence are periodic in nature. The periods tend to be natural values (once an hour, once a day, etc.).
- Manual tuning of jobs is hard: 75% of jobs are over-provisioned even at their peak. A fifth of jobs are more than 10 times over-provisioned!
- Users don't change the provisioning settings for their periodic jobs beyond the initial

setup (80% of periodic jobs saw no change in their resource provisioning).

- Run times are unpredictable — noisy neighbours (the amount of sharing) have some impact, but even when removing common sources of inherent variability (data availability, failures, network congestion), run times remain unpredictable (e.g. due to stragglers). The chart below shows the wide variability in run times of five different TPC-H queries on a 500-container system:



- Cluster hardware keeps evolving. For example, over a one-year period, the ratio between machines of type SKU1 and machines of type SKU2 changed from 80/20 to 55/45, and the total number of nodes also changed:

This is notable, because even seemingly minor hardware differences can impact job run time significantly — e.g., 40% difference in run time on SKU1 vs SKU2 for a Spark production job.

  - Jobs don't stand still: within a one-month trace of data, 15-20% of periodic jobs had at least one large code delta (more than 10% code difference) and over 50% had at least one small delta: "Even an optimal static tuning is likely going to drift out of optimality over time."

Based on their findings, the team concluded that:

1. History-based approaches can model the "normal" behaviour of a job.

2. Handling outliers without wasting resources requires a dynamic component that performs reprovisioning online.

3. While each source of variance can be addressed with an ad hoc solution, providing a general-purpose line of defence is paramount.

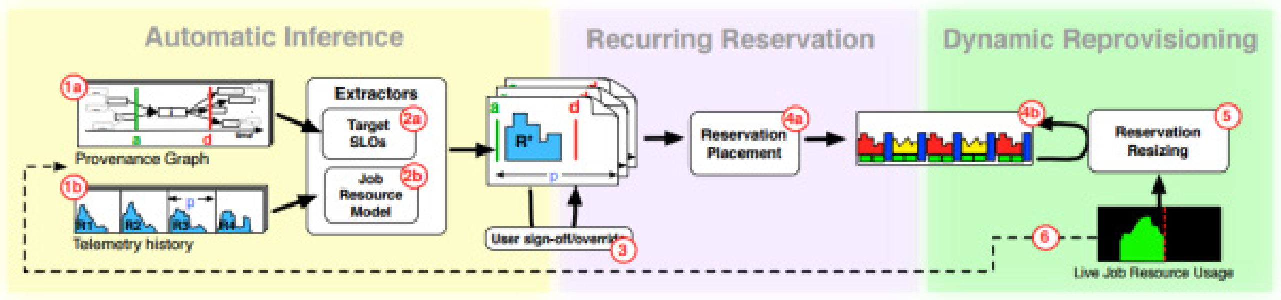


Figure 1: Conceptual view of Morpheus' achirecture. Numbers/letters match the "Life" of a periodic job

MORPHEUS OVERVIEW

Given a job that is periodically submitted by a user (with manually provisioned resources), Morpheus quietly monitors the job over time and captures:

- a provenance graph with data dependencies and ingress/egress operations and
- telemetry history in the form of resource skylines that capture resource utilisation.

Following a number of successful runs of the job, a service-level objective (SLO) inference component performs an offline analysis. Using the provenance graph, it determines a deadline for completion, the SLO, and — using the skylines — a model of the job resource demand over time.

At this point, the system is ready to take over resource provisioning for the job, but before it does so, the user signs off (or optionally overrides) the automatically generated SLO and job resource model.

(Wouldn't that be great as a user, to know that the system is going to monitor and try to honour an SLO for your jobs?)

Morpheus then uses recurring reservations for each managed job. These set aside resources over time for running the job, based on the job resource model, and each new instance of the job runs using these dedicated resources.

To address variability, a dynamic reprovisioning component monitors job progress online and adjusts the reservation in real time to mitigate the inherent execution variability.

Job runs continue to be monitored to learn and refine the SLO and job resource model over time. (Figure 1)

Let's look a little more closely at a couple of the key steps.

AGREEING ON SLOS

Petabytes of logs gathered daily across the production environments are processed to create a "semantically rich and compact (few TBs) graph representation of the raw logs."

This representation gives us a unique vantage point with nearly perfect close-world knowledge of the meaningful events in the cluster.

Periodic jobs are uncovered by looking for (templatized) job-name matches, an approximate match on source-code signatures (What is an approximate match for a signature? I'm guessing this is not a hash-based signature…), and submissions with near-constant inter-arrival time.

We say that a job has an actionable deadline if its output is consumed at an approximately fixed time, relative to the start of the period (e.g., everyday at 4 pm), and if there is a non-trivial amount of slack between the job end and the deadline.

Morpheus collects usage patterns for these jobs and determines resource skylines. It then becomes a linear-programming (LP) optimisation problem to determine the best resource allocations. The cost function uses one term that penalises for over-allocation and another term that penalises for under-allocation. (Figure 2)

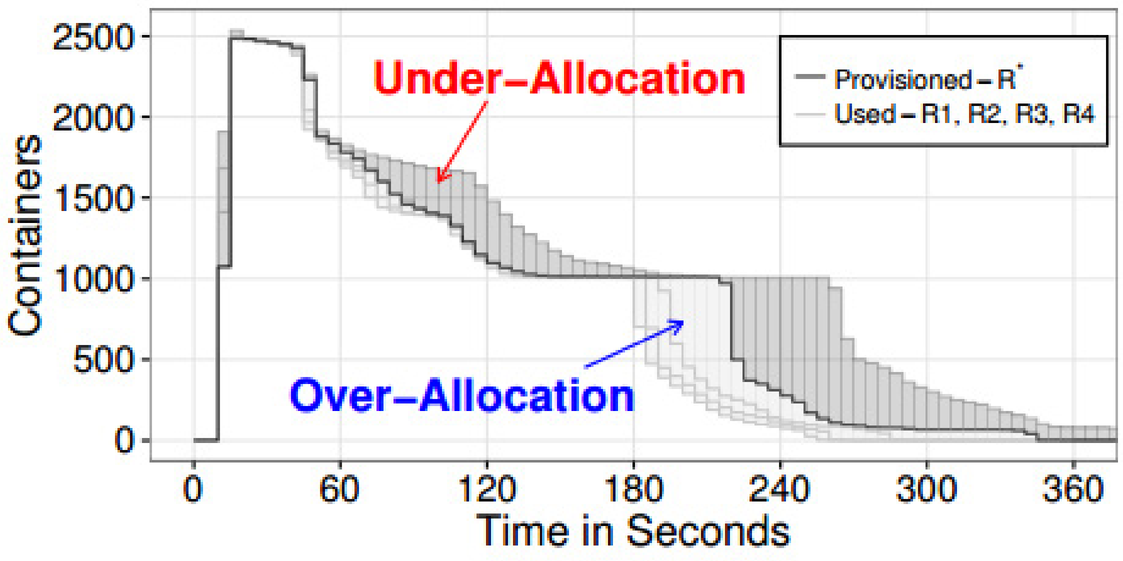


Figure 2: LP deriving a provisioned skyline R*, from four runs (R1-R4) of TPC-H Query 12 (10TB scale)

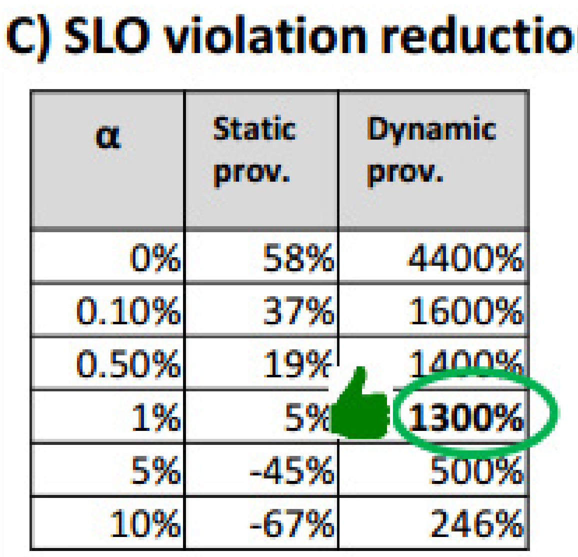
“In a simulation run from the largest dataset available to the team, Morpheus was able to automatically derive SLOs for over 70% of the millions of instances of periodic jobs in the trace.

The LP has (O(N x K)) number of variables and constraints. Our sampling granularity is typically one minute, and we keep roughly one month worth of data. This generates less than 100K variables and constraints. A state-of-the-art solver (e.g., Gurobi, CPlex) can solve an LP of millions of variables and constraints in up to few minutes. Since we are way below the computational limit of top solvers, we obtain a solution within few seconds for all periodic jobs in our clusters.

MAKE IT SO

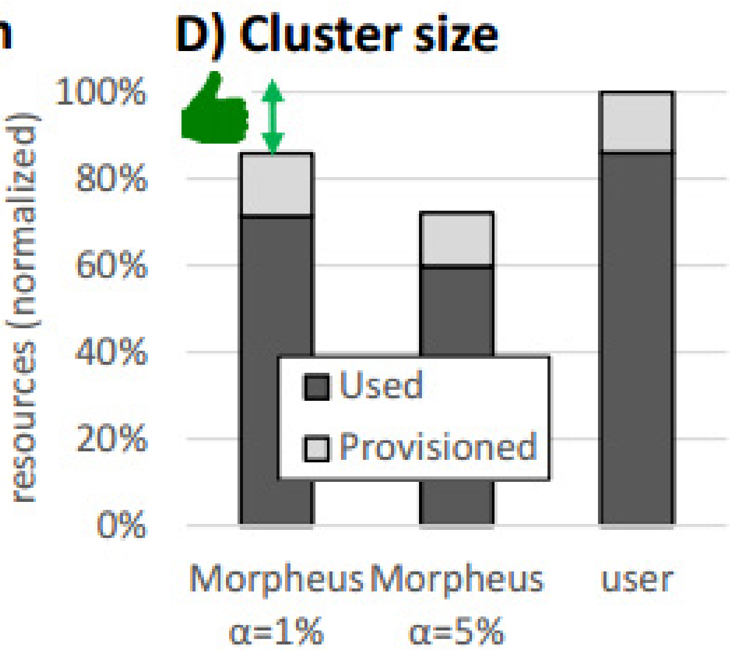
With all the requirements collected, a packing algorithm called LowCost is used to match jobs to resources. LowCost allocates containers to all periodic jobs such that their requirements are met by the deadline, and at the same time it tries to minimize waiting time for non-periodic ad hoc jobs. The overall objective of LowCost is to minimise the maximum total allocation over time. It does so following a greedy procedure that iteratively places containers at cost-efficient positions.

While reservations can eliminate sharing-induced unpredictability, they provide little protection against inherent unpredictability arising from hard-to-control exogenous causes, such as infrastructure issues (e.g., hardware replacements..., lack of isolation among tasks of multiple jobs, and framework code updates) and job-centric issues (changes in the size, skew, availability of input data, changes in code/functionalities, etc.).



C) SLO violation reduction

α	Static prov.	Dynamic prov.
0%	58%	4400%
0.10%	37%	1600%
0.50%	19%	1400%
1%	5%	1300%
5%	-45%	500%
10%	-67%	246%



The dynamic reprovisioning algorithm (DRA) deals with these issues. The DRA continuously monitors the resource consumption of a job, compares it with the resources allocated in the reservation and stretches the skyline of resources to accommodate a slower-than-expected job execution.

KEY RESULTS

In a simulation run from the largest dataset available to the team, Morpheus was able to automatically derive SLOs for over 70% of the millions of instances of periodic jobs in the trace. (The remainder don't have enough data — e.g., they are weekly jobs and so only appear four times in the one-month trace.) Compared to the current manual provisioning, Morpheus reduces worst-case SLO misses by 13 times! The packing algorithm does a good job of driving utilisation at the same time, lowering the overall cluster cost.

SLO extraction and packing contribute to lower the baseline cluster size by 6%. Job resource modelling (using the extracted skyline rather than user-supplied provisioning) lowers cluster size by a further 16%. Combined, they give a 19% reduction.

[By] turning on/off our dynamic reprovisioning we can either 1) match the user utilization level and deliver 13x lower violations, or 2) match the current SLO attainment and reduce cluster size by over 60% (since we allow more aggressive tuning of the LP, and repair underallocations dynamically).