Abstract—Researchers and practitioners studying best practices strive to design Machine Learning (ML) application systems and software that address software complexity and quality issues. Such design practices are often formalized as architecture and design patterns by encapsulating reusable solutions to common problems within given contexts. In this paper, software-engineering architecture and design (anti-)patterns for ML application systems are analyzed to bridge the gap between traditional software systems and ML application systems with respect to architecture and design. Specifically, a systematic literature review confirms that ML application systems are popular due to the promotion of artificial intelligence. We identified 32 scholarly documents and 48 gray documents out of which 38 documents discuss 33 patterns: 12 architecture patterns, 13 design patterns, and 8 anti-patterns. Additionally, a survey of developers reveals that there are 7 major architecture patterns and 5 major design patterns. Then the relationships among patterns are identified in a pattern map.

The popularity of ML techniques has increased in recent years. ML is used in many domains, including cyber security, IoT, and autonomous cars. ML techniques rely on mathematics and software engineering. The former generates algorithms, develops capabilities to learn from input data, and produces representative models. The latter is employed for implementation and performance.

Although many works investigated the mathematics and computer science on which ML techniques are built, few have examined their implementation. This situation raises many concerns. The first is software complexity of ML techniques. The second is quality of the available implementations, including performance and reliability. The third is model quality, which may be negatively impacted by software bug. These concerns could be alleviated if developers could demonstrate the software quality of their implementations. Consequently, researchers and practitioners have been studying best practices to design ML application systems to address issues with software complexity and quality of ML techniques. Such practices are often formalized as architecture patterns and design patterns. These patterns encapsulate reusable solutions to commonly occurring problems within ML application systems and software design.

Herein we report the results of a systematic
literature review (SLR) of good/bad design patterns for ML. Based on the results, we also report on developers’ perceptions as well as relationships among extracted ML patterns1.

How the Literature Addresses Software Engineering ML Design Patterns

We performed a SLR of both academic and gray literature to collect SE good (bad) design patterns for ML application systems and software. For the academic literature, we chose Engineering Village, which is a search platform that provides access to 12 engineering document databases such as Ei Compendex and Inspec. Engineering Village can search in all recognized scholarly engineering journals, conferences, and workshop proceedings with a unique search query. Moreover, Engineering Village automatically detects and removes most duplicative search results. On August 14, 2019, we designed and used the following query specifying “pattern” as well as keywords related to patterns to search for documents addressing ML design practice: ((((system) OR (software)) AND (machine learning) AND ((implementation pattern) OR (pattern) OR (architecture pattern) OR (design pattern) OR (anti-pattern) OR (recipe) OR (workflow) OR (practice) OR (issue) OR (template))) WN ALL) + ((cpx OR ins OR kna) WN DB) AND ((ca OR ja OR ip OR ch) WN DT).

For the gray literature, we used a Google search on August 16, 2019. The query was the same as that for the academic literature: (system OR software) “Machine learning” (pattern OR “implementation pattern” OR “architecture pattern” OR “design pattern” OR anti-pattern OR recipe OR workflow OR practice OR issue OR template) and “machine implementation pattern” OR architecture pattern” OR “design pattern” OR anti-pattern OR recipe OR workflow OR practice OR issue OR template.

We retrieved 32 scholarly documents and 48 gray literature documents. For each document, two of the authors vetted whether it should be included in our SLR or not. The titles and abstracts were initially reviewed. Then the entire document was read to determine whether the document pertained to software-engineering practices for ML application systems. This process identified 19 scholarly documents and 19 gray documents. All the data are available on-line2.

Figure 1 shows the trend in the number of documents related to design for ML application systems in the past decade. ML application systems have recently become popular due to the promotion of artificial intelligence. Since 2008, academic and gray documents have discussed good (bad) practices of ML application systems design.

Overview of ML Patterns

Two of the authors read half of the documents. Each author extracted patterns independently. Then the remaining author vetted each pattern. Although 69 patterns related to the architecture and design of ML application systems were initially identified, only 33 remained after the vetting process.

In general, systems and software design processes have two major phases [2] with different abstraction levels: architectural design and detailed design. Similarly, the extracted patterns can be classified into two types: ML architecture patterns and ML design patterns. Documents describing ML architecture patterns recommend architecture designs of ML application systems and software to address recurrent architectural problems such as ensuring maintainability of ML components. In contrast, ML design patterns address recurrent detailed design problems such as enabling proper communications among specific modules.

1 Preliminary results of our SLR is presented at [1]. In this paper, we examined all patterns and relationships among patterns in detail. We also newly studied developers’ perceptions on patterns.

2 http://www.washi.cs.waseda.ac.jp/ml-patterns/
<table>
<thead>
<tr>
<th>Pattern Name</th>
<th>Problem (excerpt)</th>
<th>Solution (excerpt)</th>
<th>NP and Source</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data Lake</td>
<td>We cannot foresee the kind of analyses that will be performed on the data and which frameworks will be used to perform these analyses.</td>
<td>The data ranging from structured data to unstructured data should be stored as “raw” as possible and the centralized data repository should allow parallel analyses of different kinds and with different frameworks.</td>
<td>5, [4], <a href="http://bit.ly/33DTKTe">http://bit.ly/33DTKTe</a></td>
</tr>
<tr>
<td>Distinguish Business Logic from ML Models</td>
<td>The overall business logic should be isolated as much as possible from the ML models so that they can be changed/overridden when necessary without impacting the rest of the business logic.</td>
<td>Separate the business logic and the inference engine, loosely coupling the business logic and ML-specific dataflows.</td>
<td>4, [5]</td>
</tr>
<tr>
<td>Microservice Architecture</td>
<td>ML applications may be confined to some “known” ML frameworks and miss opportunities for more appropriate frameworks.</td>
<td>Data scientists working with or providing ML frameworks can make these frameworks available through microservices.</td>
<td>4, [4], <a href="http://bit.ly/2DyHGrV">http://bit.ly/2DyHGrV</a></td>
</tr>
<tr>
<td>Data-Algorithm-Serving-Evaluator</td>
<td>Prediction systems should connect different pieces in the data processing pipeline into one coherent system and prototyping predictive model.</td>
<td>Separate the following like MVC for ML data (data source and data preparator), algorithm(s), serving, and evaluator.</td>
<td>2, [3], <a href="http://bit.ly/2r6edmnu">http://bit.ly/2r6edmnu</a></td>
</tr>
<tr>
<td>Event-driven ML Microservices</td>
<td>Due to frequent prototyping of ML models and constant changes, development teams must be agile to build, deploy, and maintain complex data pipelines.</td>
<td>Construct pipelines by chaining together multiple microservices, each of which listens for the arrival of some data and performs its designated task.</td>
<td>2, [2], <a href="http://bit.ly/3DZnuxH">http://bit.ly/3DZnuxH</a></td>
</tr>
<tr>
<td>Lambda Architecture</td>
<td>Real-time data processing requires scalability, fault tolerance, predictability, and other qualities. It must be extensible.</td>
<td>The batch layer keeps producing views at every set batch interval while the speed layer creates the relevant real-time/speed views. The serving layer orchestrates the query by querying both the batch and speed layer, merges it.</td>
<td>2, [4], <a href="http://bit.ly/33DTKTe">http://bit.ly/33DTKTe</a></td>
</tr>
<tr>
<td>Parameter-Server Abstraction</td>
<td>For distributed learning, widely accepted abstractions are lacking.</td>
<td>Distribute both data and workloads over worker nodes, while the server nodes maintain globally shared parameters, which are represented as vectors and matrices.</td>
<td>2, [4]</td>
</tr>
<tr>
<td>Daisy Architecture</td>
<td>The ability to scale content production processes must be acquired via the use of ML. Then the coverage of that tooling must be extended over as much of their remaining content.</td>
<td>Utilize Kanban, scaling, and microservice to realize pull-based, automated, on-demand, and iterative processes.</td>
<td>1, [3], <a href="http://bit.ly/2DyHGrV">http://bit.ly/2DyHGrV</a></td>
</tr>
<tr>
<td>Gateway Routing Architecture</td>
<td>When a client uses multiple services, it can be difficult to set up and manage individual endpoints for each service.</td>
<td>Install a gateway before a set of applications, services, or deployments and use application layer routing requests to the appropriate instance.</td>
<td>1, [4], <a href="http://bit.ly/37Xkguc">http://bit.ly/37Xkguc</a></td>
</tr>
<tr>
<td>Kappa Architecture</td>
<td>It is necessary to deal with huge amount of data with less code resource.</td>
<td>Support both real-time data processing and continuous reprocessing with a single stream processing engine.</td>
<td>1, [5], <a href="http://bit.ly/33DTKTe">http://bit.ly/33DTKTe</a></td>
</tr>
<tr>
<td>Closed-Loop Intelligence</td>
<td>It is necessary to address big, open-ended, time-changing or intrinsically hard problems.</td>
<td>Connect machine learning to the user and close the loop. Design clear interactions along with implicit and direct outputs.</td>
<td>0, [4], <a href="http://bit.ly/2L8ZpdB">http://bit.ly/2L8ZpdB</a></td>
</tr>
<tr>
<td>Federated Learning</td>
<td>Standard machine learning approaches require centralizing the training data on one machine or in a datacenter.</td>
<td>Employ Federated Learning, which enables mobile phones to collaboratively learn a shared prediction model while keeping all the training data on the device.</td>
<td>0, [3], <a href="http://bit.ly/2qaRJk3">http://bit.ly/2qaRJk3</a></td>
</tr>
</tbody>
</table>

In addition to the ML architecture patterns and design patterns, we also identified ML anti-patterns. Similar to general anti-patterns [5], ML anti-patterns (including code smells and technical debts [6]) describe commonly occurring situations that generate negative consequences in ML application systems and software design.

Tables 1, 2 and 3 list the extracted ML patterns. Of these, 18 (55%) were extracted from the scholarly documents, while 15 (45%) were from the gray documents. Twelve (36%) are ML architecture patterns, and thirteen (39%) are ML design patterns, suggesting that the unique nature of design in ML application systems and software appears at both of the architecture level and the detailed design level. The remaining are 8 ML anti-patterns (24%).

### How Engineers’ Perceived Software Engineering ML Design Patterns

ML techniques are concrete solutions to practical problems. Hence, ML developers may already have built a body of knowledge on the good (bad) design practices of ML development. To clarify how ML developers perceive existing ML patterns, we asked 19 developers and researchers...
at Japanese companies and research organizations to complete a survey during a workshop in October 2019. After a brief introduction of all patterns, we inquired on whether or not they used any of the ML architecture and design patterns.

As shown in Table 1, multiple participants used the seven major ML architecture patterns (out of 12): “Data-Algorithm-Serving-Evaluator”, “Data Lake”, “Distinguish Business Logic from ML Models”, “Microservice Architecture”, “Event-driven ML Microservices”, “Lambda Architecture”, and “Parameter-Server Abstraction”. And, all ML architecture patterns except for “Federated Learning” and “Closed-Loop Intelligence” are used at least by one participant.
Table 3. Extracted ML Anti-Patterns

<table>
<thead>
<tr>
<th>Pattern Name</th>
<th>Problem (excerpt)</th>
<th>Source</th>
</tr>
</thead>
<tbody>
<tr>
<td>Big Ass Script Architecture</td>
<td>When all code is placed in one big ass script, it becomes difficult to reuse in future analysis, understand how it works, and debug.</td>
<td><a href="http://bit.ly/35QPb9N">http://bit.ly/35QPb9N</a></td>
</tr>
<tr>
<td>Abstraction Debt</td>
<td>For distributed learning, widely accepted abstractions are lacking.</td>
<td>[4]</td>
</tr>
<tr>
<td>Dead Experimental Codepaths</td>
<td>The code-paths accumulated by individual changing can create a growing debt due to the increasing difficulties of maintaining backward compatibility.</td>
<td>[4]</td>
</tr>
<tr>
<td>Glue Code</td>
<td>Glue code is costly in the long term because it tends to freeze a system to the peculiarities of a specific package.</td>
<td>[4]</td>
</tr>
<tr>
<td>Multiple-Language Smell</td>
<td>Using multiple languages increases the cost of effective testing and can increase the difficulty of transferring ownership to other individuals.</td>
<td>[4]</td>
</tr>
<tr>
<td>Pipeline Jungles</td>
<td>The system to prepare data in an ML-friendly format may become a pipeline jungle, and managing these pipelines is difficult and costly.</td>
<td>[4]</td>
</tr>
<tr>
<td>Plain-Old-Data Type Smell</td>
<td>The rich information used and produced by ML systems is often encoded with plain data types like raw floats and integers.</td>
<td>[4]</td>
</tr>
<tr>
<td>Undeclared Consumers</td>
<td>Undeclared consumers are dangerous because they create a hidden tight coupling of model MA to other parts of the stack.</td>
<td>[4]</td>
</tr>
</tbody>
</table>

In contrast, multiple participants used the five major ML design patterns (out of 13 in Table 2): “Handshake”, “Isolate and Validate Output of Model”, “ML Versioning”, “Test Infrastructure Independently from ML”, and “Wrap Black-Box Packages into Common APIs”. In addition, none of the participants reported using the three ML design patterns, suggesting that ML design patterns are less accepted in comparison to ML architecture patterns.

Example of Major Software Engineering ML Pattern

Here, we describe one major ML architecture pattern and its usage. We selected “Distinguish Business Logic from ML Model” (originally named as “Multi-Layer Architectural Pattern” [3])

Pattern Name

Distinguish Business Logic from ML Model

Intent

Isolate failures between business logic and ML learning layer to help developers debug ML application systems easily.

Also Known As

Machine Learning System Architectural Pattern for Improving Operational Stability.

Problem

ML application systems are complex because their ML components must be (re)trained regularly and have an intrinsic non-deterministic behavior. Similar to other systems, the business requirements for these systems and the ML algorithms change over time.

Solution

Define clear APIs between the traditional and ML components. Place the business and ML components with different responsibilities into three layers (Fig. 2). Divide data flows into three.

Applicability

It is applicable to any ML application system with outputs that depend on ML techniques.

Consequences

Decoupling “traditional” business and ML components allows the ML components to be monitored and adjusted to meet users’ requirements and changing inputs.
Usage Example
Figure 3 presents an example of implementation of the pattern “Distinguish Business Logic from ML Model” in a Slack-based Chatbot system. By referring to the architecture pattern, we easily specified necessary elements as well as relationships among them while having clear separation between the Chatbot service (as the business logic) and the underlying ML components.

ML Pattern Map
To help developers navigate the patterns, we identified the following four types of relationships among the patterns using basic relation types [10]: X is similar to Y but has different objectives, X can use Y in its solution, X and Y can be combined to solve larger problems, and, X can mitigate the problem of Y.

Figure 4 shows a result of identification of the relationships among ML patterns as a pattern map. For example, “Closed-Loop Intelligence” is an architecture pattern to have clear interactions with users; it would mitigate the problem of “Undeclared Customers”. “Closed-Loop Intelligence” can be combined with other high-level architecture patterns that address business logic and user interactions such as “Distinguish Business Logic from ML Models” and “Data-Algorithm-Serving-Evaluator”. “Gateway Routing Architecture” is similar to “Distinguish Business Logic from ML Models” since both use “Data Lake” in their solutions to handle variety of data; however, objectives of these two architecture patterns are quite different.

According to the number of connected relationships, the fundamental patterns are “Big Ass Script Architecture” and “Separation of Concerns and Modularization of ML Components”, suggesting that developers should initially identify the corresponding problems in their design and solve them by referring to these patterns as well as connected related patterns.

In terms of the anti-patterns, we suggest that developers should refactor their code as soon as these symptoms appear by applying corresponding design patterns connected in the pattern map.

CONCLUSION
To bridge the gap between traditional software systems and ML application systems with respect
to architecture and design, software-engineering architectural and design (anti-)patterns for ML application systems were analyzed via an SLR and a survey of developers. ML application systems are quite popular due to the recent promotion of artificial intelligence. From the 32 scholarly documents and 48 gray documents identified in the SLR, 12 ML architecture patterns, 13 ML design patterns, and 8 ML anti-patterns were identified. A survey of developers reveals that there are 7 major ML architecture patterns and 5 major ML design patterns. The relationships among the patterns were elucidated.

In the future, we plan to write all patterns into a standardized format because not all the identified patterns are well written. Additionally, we intend to investigate the impact of SE patterns on the quality attributes of ML application systems. These patterns will be validated by not only sharing these patterns to participants at the Writers’ Workshop at the PLoP conference series but also by contacting the original designer. We also plan to involve more practitioners from the community by continuing the survey and the SLR.

ACKNOWLEDGMENT

The authors would like to thank Prof. Naoshi Uchihiara, Mr. Norihiko Ishitani, Dr. Takuo Doi, Dr. Shunichiro Suenaga, Mr. Yasuhiro Watanabe and Prof. Kazunori Sakamoto for their helps. This work was supported by JST-Mirai Program Grant Number JP18077318, Japan.

REFERENCES


