A DEMOGRAPHICS SLR

See Figures 3, 4, 5.

Figure 3: Distribution of manually inspected and used articles grouped by source.

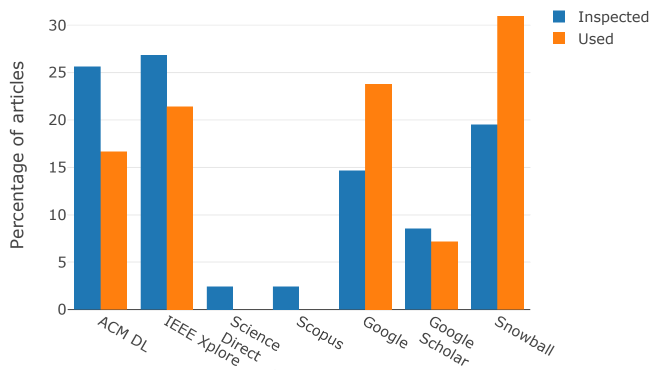


Figure 4: Distribution of manually inspected and used articles grouped by publication year.

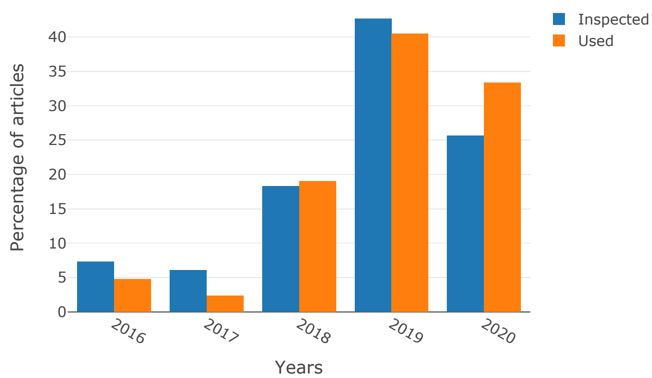
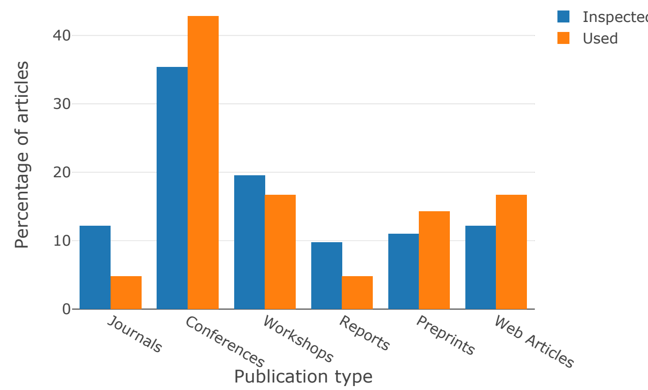


Figure 5: Distribution of manually inspected and used articles grouped by publication type.



B LIST OF ARTICLES SLR

See Table 6.

Source	Title	Used
ACM DL	A framework for managing uncertainty in software architecture	0
ACM DL	A Report on the First Workshop on Software Engineering for Artificial Intelligence (SE4AI 2020)	0
ACM DL	Achieving guidance in applied machine learning through software engineering techniques	1
ACM DL	Deep learning UI design patterns of mobile apps	0
ACM DL	Designing the Software Systems of the Future	1
ACM DL	Do you want to become an AI and machine learning software engineer?	0
ACM DL	Does fixing bug increase robustness in deep learning?	0
ACM DL	Emerging and Changing Tasks in the Development Process for Machine Learning Systems	1
ACM DL	Hacking Machine Learning	0
ACM DL	Intelligent Software Engineering: Synergy Between AI and Software Engineering	0
ACM DL	Keeping intelligence under control	0
ACM DL	Robustness testing of autonomy software	0
ACM DL	Software Engineering for distributed autonomous real-time systems	0
ACM DL	Software Engineering for Machine Learning: A Case Study	1
ACM DL	Taxonomy of real faults in deep learning systems	0
ACM DL	Teaching software engineering for AI-enabled systems	0
ACM DL	Toward a holistic software systems engineering approach for dependable autonomous systems	1
ACM DL	Towards classes of architectural dependability assurance for machine-learning-based systems	1
ACM DL	Tutorial on Software Testing & Quality Assurance for Machine Learning Applications	0
ACM DL	Self-organizing infrastructure for machine (deep) learning at scale	0
ACM DL	Sensemaking Practices in the Everyday Work of AI/ML Software Engineering	1
IEEE Xplore	A Bird's Eye View on Requirements Engineering and Machine Learning	0
IEEE Xplore	A detailed survey of Artificial Intelligence and Software Engineering: Emergent Issues	0
IEEE Xplore	A Safe, Secure, and Predictable Software Architecture for Deep Learning in Safety-Critical Systems	0
IEEE Xplore	A survey of software quality for machine learning applications	0
IEEE Xplore	AI Safety Landscape From short-term specific system engineering to long-term artificial general intelligence	0
IEEE Xplore	Analysis of Software Engineering for Agile Machine Learning Projects	0
IEEE Xplore	Can AI close the design-code abstraction gap?	0
IEEE Xplore	Deep learning development review	0
IEEE Xplore	Designing Safety Critical Software Systems to Manage Inherent Uncertainty	1
IEEE Xplore	How Do Engineers Perceive Difficulties in Engineering of Machine-Learning Systems? - Questionnaire Survey	1
IEEE Xplore	How does Machine Learning Change Software Development Practices?	1
IEEE Xplore	Improved Self-Management Architecture in Self-Adaptive System	0
IEEE Xplore	What is AI software testing? And why?	0
IEEE Xplore	Requirements engineering challenges in building AI-based complex systems	1
IEEE Xplore	Security engineering for machine learning	1
IEEE Xplore	Software engineering challenges of deep learning	1
IEEE Xplore	Software Engineering for Machine-Learning Applications: The Road Ahead	0
IEEE Xplore	Studying Software Engineering Patterns for Designing Machine Learning Systems	1
IEEE Xplore	Testing and Quality Validation for AI Software–Perspectives, Issues, and Practices	0
IEEE Xplore	Towards concept based software engineering for intelligent agents	0
IEEE Xplore	Uncertain requirements, assurance and machine learning	1
IEEE Xplore	Understanding Development Process of Machine Learning Systems: Challenges and Solutions	1
Science Direct	Assessing the drivers of machine learning business value	0
Science Direct	AI service system development using enterprise architecture modeling	0
Scopus	Big Data Analytics in Building the Competitive Intelligence of Organizations	0
Scopus	Complex, Intensive Systems and Software Intelligent	0
Google Scholar	A test architecture for machine learning product	1
Google Scholar	An Analysis of ISO 26262: Using Machine Learning Safely in Automotive Software	0
Google Scholar	An overview of next-generation architectures for machine learning: Roadmap, opportunities and challenges in the IoT era	0

Google Scholar	SNaP ML: A hierarchical framework for machine learning	0
Google Scholar	Software Architecture Design of the Real-Time Processes Monitoring Platform	1
Google Scholar	Software Architecture in a Changing World	1
Google Scholar	Solution Patterns for Machine Learning	0
Google	A Taxonomy of Software Engineering Challenges for Machine Learning Systems: An Empirical Investigation	1
Google	Continuous Delivery for Machine Learning	1
Google	Demystifying Data Lake Architecture	1
Google	Deploy, Connect and Execute Scientific Models	0
Google	Ethics guidelines for trustworthy AI	1
Google	Hidden technical debt in machine learning systems	1
Google	The National Artificial Intelligence Research and Development Strategic Plan: 2019 Update	1
Google	ML Reference Architecture	1
Google	Machine Learning Architecture and Design Patterns	1
Google	Method for Assessing the Applicability of AI Service Systems	0
Google	Requirements for Trustworthy Artificial Intelligence – A Review	1
Google	Software Engineering Practice in the Development of Deep Learning Applications	1
Snowball	A Design Pattern for Machine Learning with Scala, Spray and Spark	1
Snowball	AI Engineering: 11 Foundational Practices	1
Snowball	Adoption and Effects of Software Engineering Best Practices in Machine Learning	1
Snowball	ClearTK 2.0: Design patterns for machine learning in UIMA	0
Snowball	Continuous Training for Production ML in the TensorFlow Extended (TFX) Platform	1
Snowball	Data Validation for Machine Learning	1
Snowball	Daisy Architecture	1
Snowball	Expanding AI's impact with organisational learning	0
Snowball	Machine learning at Facebook: Understanding inference at the edge	1
Snowball	Machine Learning Software Engineering in Practice: An Industrial Case Study	1
Snowball	Machine Learning System Architectural Pattern for Improving Operational Stability	1
Snowball	Patterns (and Anti-Patterns) for Developing Machine Learning Systems	1
Snowball	Rules of Machine Learning	1
Snowball	Towards using probabilistic models to design software systems with inherent uncertainty	1
Snowball	Scaling distributed machine learning with the parameter server	0
Snowball	Scaling Machine Learning as a Service	1

Table 6: Manually inspected and used articles.

C DATA EXTRACTION SLR

See Table 7.

Table 7: Data items extracted from each article.

ID	Data Item	Description	RQ
1	Title	The title of the article.	Demographics
2	Year	The publication year.	Demographics
3	Venue	The publication venue name.	Demographics
4	Context	Academic or Industry.	Demographics
5	Source	Retrieval source.	Demographics
6	Research type	Type of research – e.g., validation research, evaluation research, opinion article.	Demographics.
7	Challenges	Documents the challenges reported in (re-) designing software with ML components.	RQ1
8	Tactics, Practices or Patterns	Documents the tactics, practices or patterns reported to meet challenges in (re-) designing software with ML components.	RQ2
9	Data type	The data type used in ML.	Data

D SOLUTIONS EXTRACTED FROM THE SLR

See Table 8.

Table 8: List of SA challenges for ML and related solutions as extracted from the SLR.

Nr.	Category	Challenges	Solutions	References
1	Reqs.	At design time the information available is insufficient to understand the customers or the projects.	Measure and document uncertainty sources.	[10, 16, 29, 39, 40, 66]
2	Reqs.	ML components lack functional requirements.	Use metrics as functional requirements. Include understandability and explainability of the outputs.	[10, 16, 19, 29, 40, 66]
3	Reqs.	ML projects have regulatory restrictions and may be subject to audits.	Analyse regulatory constraints up-front. Adopt an AI code of conduct. Design audit trails.	[23, 32, 47, 63]
4	Data	Data preparation may result in a jungle of scrapes, joins, and sampling steps, often with intermediate outputs.	Design separate modules/services for data collection and data preparation.	[19, 38, 58]
5	Data	Data quality is hard to test, and may have unexpected consequences.	Design separate modules/services for data quality assessment.	[19, 38, 43, 52, 75]
6	Design	Separate concerns between training, testing, and serving, but reuse code between them.	Standardise model interfaces.	[2, 72, 76]
7	Design	Distinguish failures between ML components and other business logic.	Separate business logic from ML components.	[53, 73]
8	Design	ML components are highly coupled, and errors can have cascading effects.	Design independent modules/services for ML and data. Relax coupling heuristics between ML and data.	[27, 49, 66]
9	Design	ML components bring inherent uncertainty to a system.	Design and monitor uncertainty metrics.	[3, 27, 49, 62, 64]
10	Design	ML components can fail silently. These failures can be hard to detect, isolate and solve.	Use metric monitoring and alerts to detect failures.	[11, 64, 71]
11	Design	ML components are intrinsically opaque, and deductive reasoning from the architecture artefacts, code or metadata is not effective.	Instrument the system to the fullest extent. Design log modules to aggregate/visualise metrics.	[27, 49, 57, 76]
12	Design	Avoid unstructured components which link frameworks or APIs (e.g., glue code).	Wrap components in APIs/modules/services.	[58]
13	Design	Automation and understanding of ML tasks is difficult (AutoML).	Version configuration files. Design the log and versioning systems to support AutoML data retrieval.	[38, 55, 63, 66, 72]
14	Testing	ML testing goes beyond programming bugs to issues that arise from model, data errors, or uncertainty.	Design model and data tests. Use CI/CD.	[2, 4, 48, 54, 75]
15	Testing	Validation of ML components for production is difficult.	Use metrics and CI/CD for validation. Use alerts, visualisations.	[56]
16	Ops.	ML components require continuous maintenance, retraining and evolution.	Design for automatic continuous retraining. Use CI/CD. Use automatic rollback.	[8, 39, 49, 56, 66, 68, 75]
17	Ops.	Manage the dependencies and consumers of ML applications.	Use authentication and access control. Log consumers of ML components.	[7, 22, 27, 58, 73]
18	Ops.	Balance latency, throughput, and fault-tolerance, needed for training and serving.	Design for batch processing (training) and stream processing (serving), i.e., lambda architecture. Physically isolate the workloads.	[15, 38, 45, 67, 72]

E DATA INTERVIEWS

See Figures 6, 7, 8.

Figure 6: Distribution of data types used by interview participants.

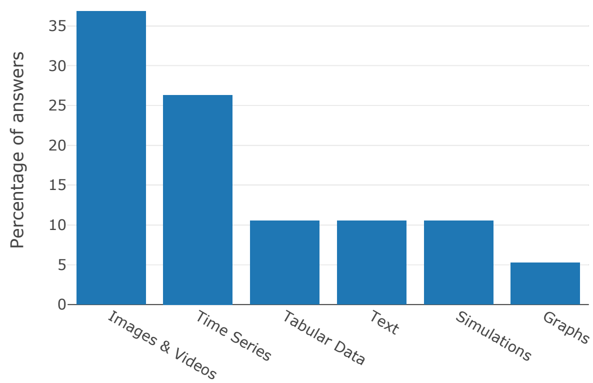


Figure 7: Distribution of architectural decision drivers from interviews.

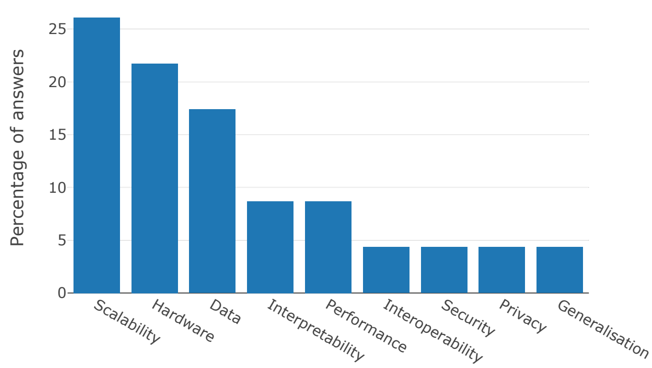
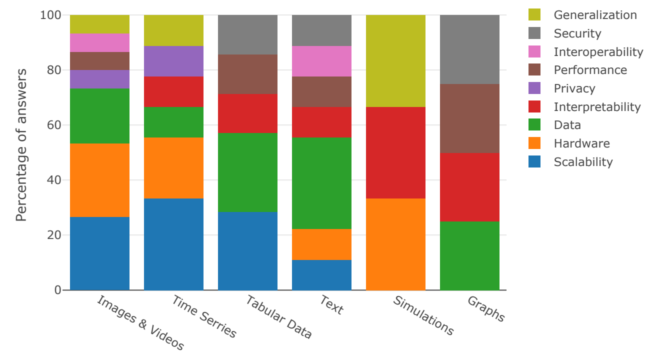


Figure 8: Distribution of architectural decision drivers grouped by data type.



F THEMES EXTRACTED FROM INTERVIEWS

Figure 9: Themes extracted for challenge 1.

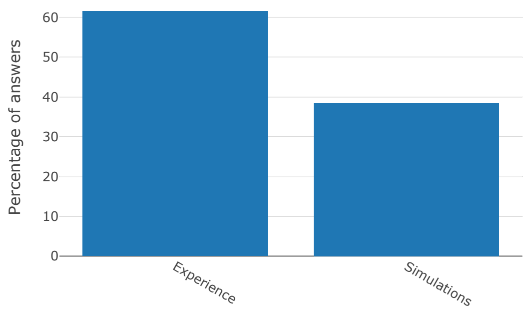


Figure 10: Themes extracted for challenge 2.

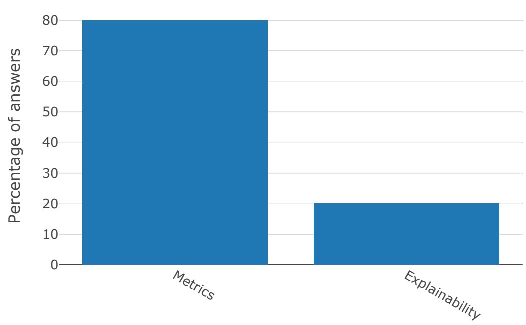


Figure 11: Themes extracted for challenge 4.

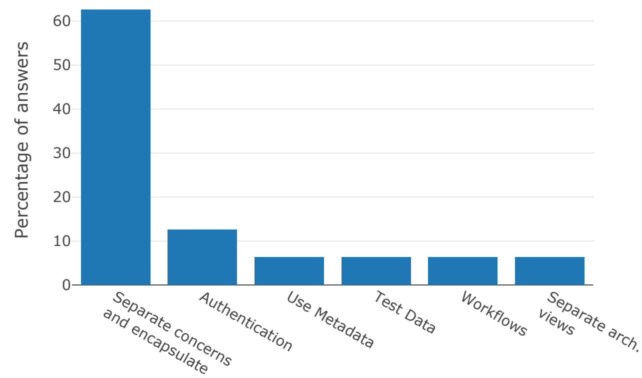


Figure 12: Themes extracted for challenge 5.

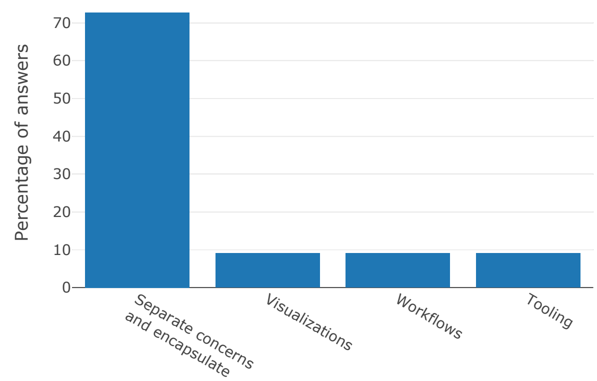


Figure 13: Themes extracted for challenge 6.

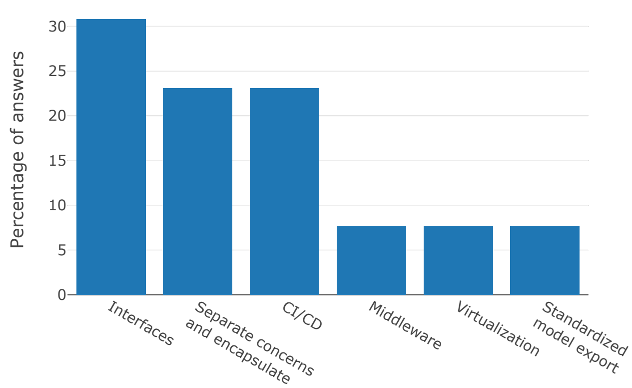


Figure 14: Themes extracted for challenge 7.

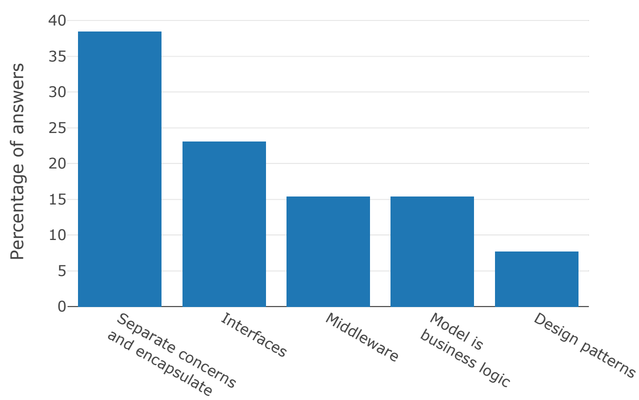


Figure 15: Themes extracted for challenge 8.

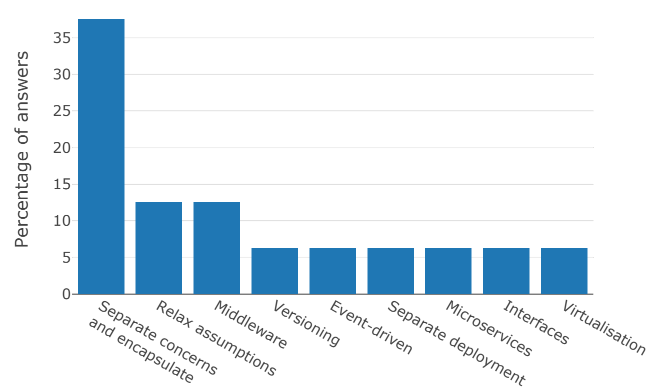


Figure 16: Themes extracted for challenge 9.

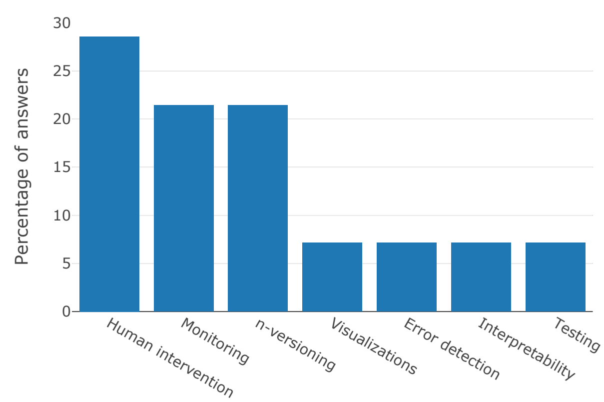


Figure 17: Themes extracted for challenge 10.

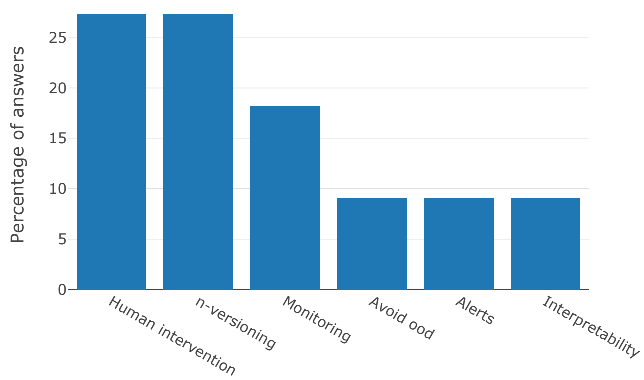


Figure 18: Themes extracted for challenge 11.

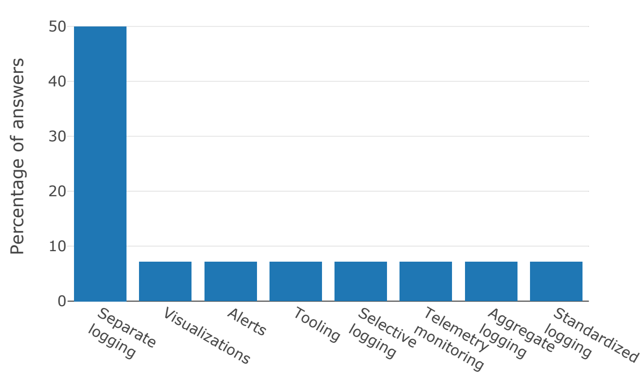


Figure 19: Themes extracted for challenge 12.

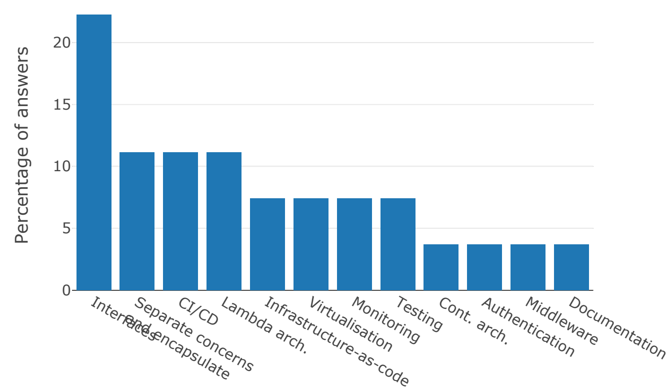


Figure 20: Themes extracted for challenge 13.

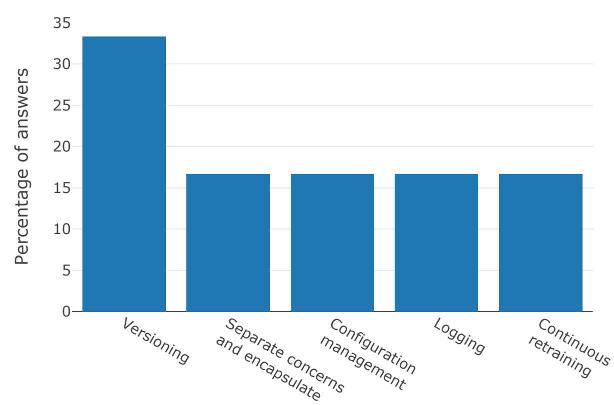


Figure 21: Themes extracted for challenge 14.

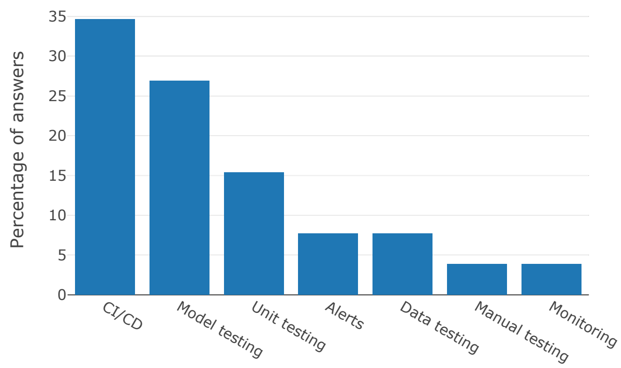


Figure 22: Themes extracted for challenge 15.

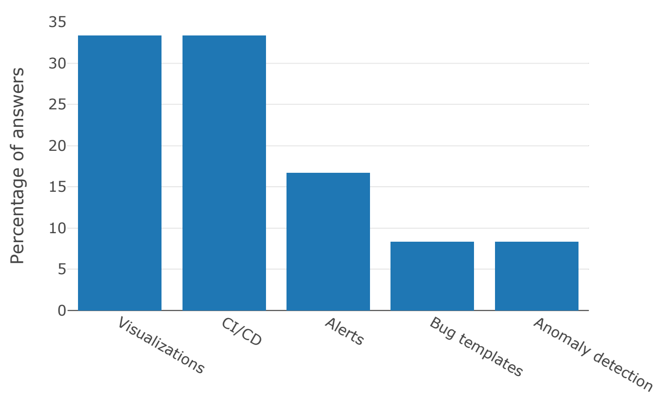


Figure 23: Themes extracted for challenge 16.

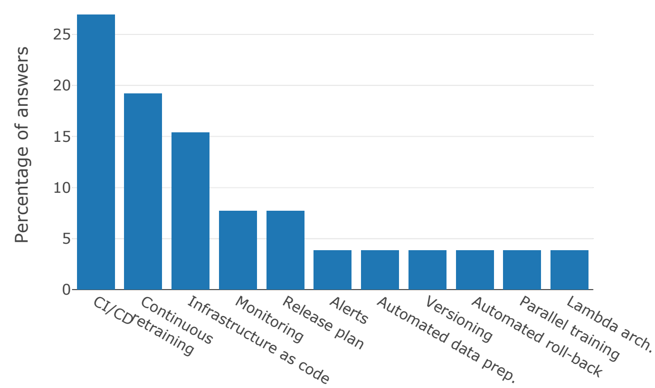


Figure 24: Themes extracted for challenge 17.

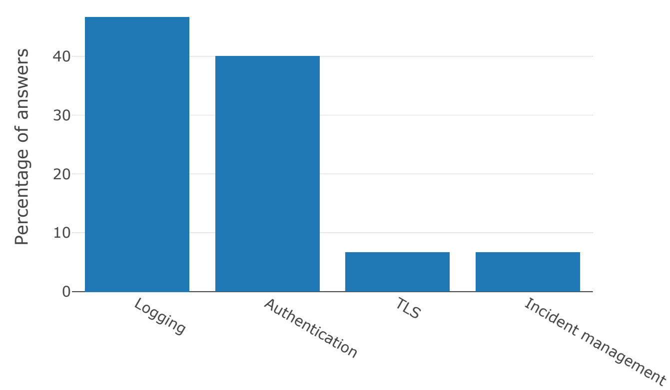
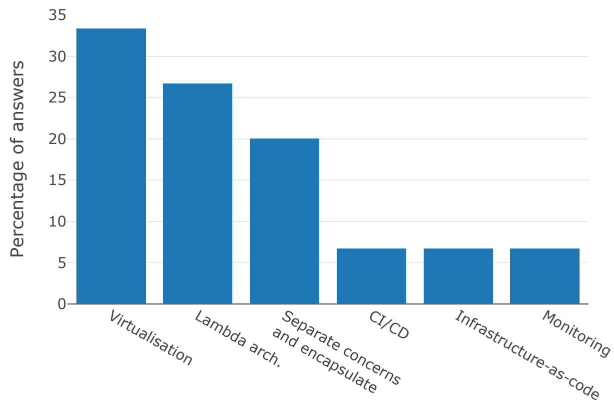


Figure 25: Themes extracted for challenge 18.



G SOLUTIONS AFTER THE INTERVIEWS

See Table 9.

Table 9: List of SA challenges and solutions after the interviews.

Nr.	Category	Challenges	Solutions	References
1	Reqs.	At design time the information available is insufficient to understand the customers or the projects.	Run simulations to gather data. Use past experience. Measure and document uncertainty sources.	[10, 16, 29, 39, 40, 66]
2	Reqs.	ML components lack functional requirements.	Use metrics as functional requirements. Include understandability and explainability of the outputs.	[10, 16, 19, 29, 40, 66]
3	Reqs.	ML projects have regulatory restrictions and may be subject to audits.	Analyse regulatory constraints up-front. Adopt an AI code of conduct. Design audit trails.	[23, 32, 47, 63]
4	Data	Data preparation may result in a jungle of scrapes, joins, and sampling steps, often with intermediate outputs.	Design separate modules/services for data collection and data preparation. Integrate external tools.	[19, 38, 58]
5	Data	Data quality is hard to test, and may have unexpected consequences.	Design separate modules/services for data quality assessment. Integrate external tools.	[19, 38, 43, 52, 75]
6	Design	Separate concerns between training, testing, and serving, but reuse code between them.	Standardise model interfaces. Use one middleware. Reuse virtualisation, infrastructure and test scripts.	[2, 72, 76]
7	Design	Distinguish failures between ML components and other business logic.	Separate business logic from ML components. Standardise interfaces and use one middleware between them.	[53, 73]
8	Design	ML components are highly coupled, and errors can have cascading effects.	Design independent modules/services for ML and data. Standardise interfaces and use one middleware. Relax coupling heuristics between ML and data.	[27, 49, 66]
9	Design	ML components bring inherent uncertainty to a system.	Use n-versioning. Design and monitor uncertainty metrics. Employ interpretable models/human intervention. Use self adaptation.	[3, 27, 49, 62, 64]
10	Design	ML components can fail silently. These failures can be hard to detect, isolate and solve.	Use metric monitoring and alerts to detect failures. Use n-versioning. Employ interpretable models. Detect out of distrribution data.	[11, 64, 71]
11	Design	ML components are intrinsically opaque, and deductive reasoning from the architecture artefacts, code or metadata is not effective.	Instrument the system to the fullest extent. Use n-versioning. Employ interpretable models. Design log modules to aggregate/visualise metrics.	[27, 49, 57, 76]
12	Design	Avoid unstructured components which link frameworks or APIs (e.g., glue code).	Wrap components in APIs/modules/services. Use standard interfaces and one middleware. Use virtualisation.	[58]
13	Design	Automation and understanding of ML tasks is difficult (AutoML).	Version configuration files. Design the log and versioning systems to support AutoML data retrieval.	[38, 55, 63, 66, 72]
14	Testing	ML testing goes beyond programming bugs to issues that arise from model, data errors, or uncertainty.	Design model and data tests. Use CI/CD. Use integration and unit tests. Use data ownership for test modules. Use manual inspection.	[2, 4, 48, 54, 75]
15	Testing	Validation of ML components for production is difficult.	Use metrics and CI/CD for validation. Use alerts, visualisations, human intervention. Design release processes.	[56]
16	Ops.	ML components require continuous maintenance, retraining and evolution.	Design for automatic continuous retraining. Use CI/CD. Use automatic rollback. Use infrastructure-as-code. Adopt standard release processes.	[8, 39, 49, 56, 66, 68, 75]
17	Ops.	Manage the dependencies and consumers of ML applications.	Encapsulate ML components in identifiable modules/services. Use authentication and access control. Log consumers of ML components.	[7, 22, 27, 58, 73]
18	Ops.	Balance latency, throughput, and fault-tolerance, needed for training and serving.	Design for batch processing (training) and stream processing (serving), i.e., lambda architecture. Physically isolate the workloads. Use virtualisation.	[15, 38, 45, 67, 72]
19	Ops.	Trace back decisions to models, data and reproduce past results.	Design for traceability and reproducibility; log pointers to versioned artefacts, version configurations, models and data.	P10
20	Org.	ML applications use heterogeneous technology stacks which require diverse backgrounds and skills.	Form multi-disciplinary teams. Adopt an AI code of conduct. Define processes for decision-making. Raise awareness about ML risks within the team.	P1

H SURVEY DEMOGRAPHICS

Figure 26: Organisation type for survey participants.

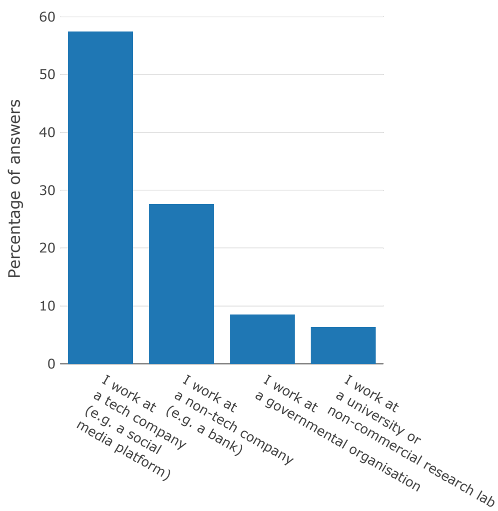


Figure 27: Experience of survey participants.

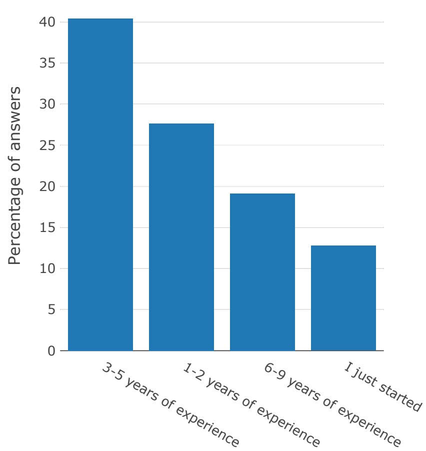


Figure 28: Team size for survey participants.

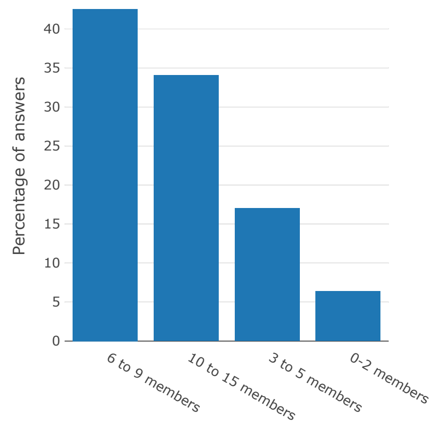


Figure 29: Data types used by survey participants.

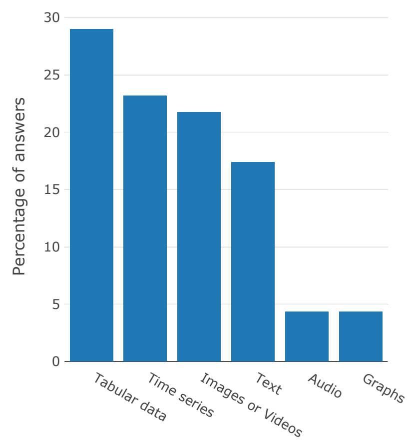


Figure 30: Deployment interval for survey participants.

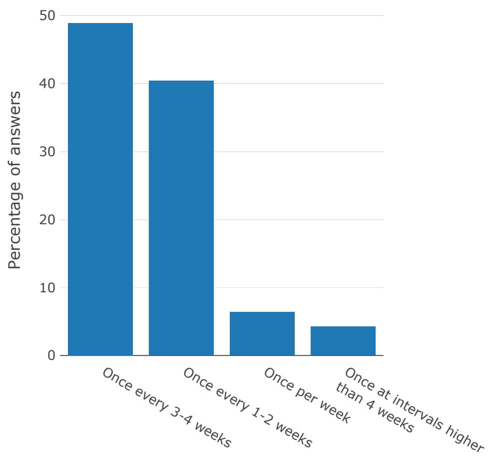
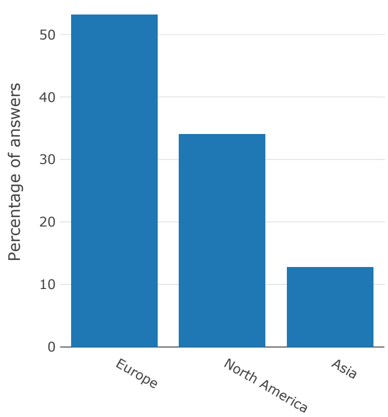


Figure 31: Regions for survey participants.



I SOLUTIONS FROM SURVEY

See Figure 32 - 51.

Figure 32: Survey solutions to practice 1.

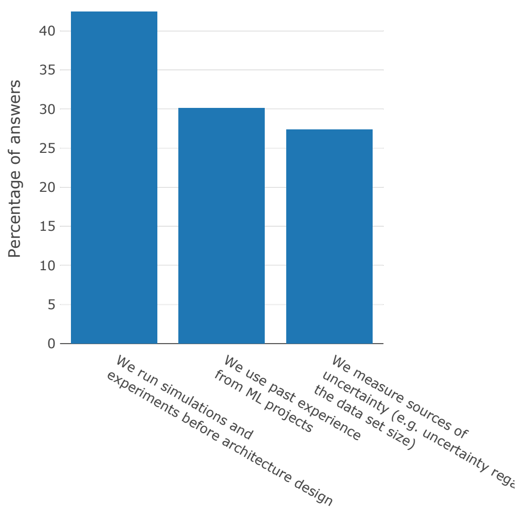


Figure 33: Survey solutions to practice 2.

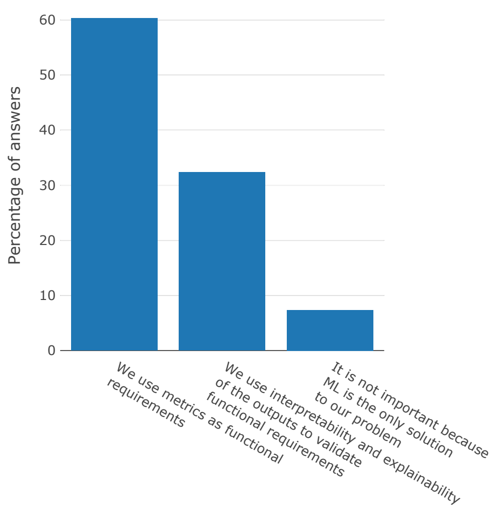


Figure 34: Survey solutions to practice 3.

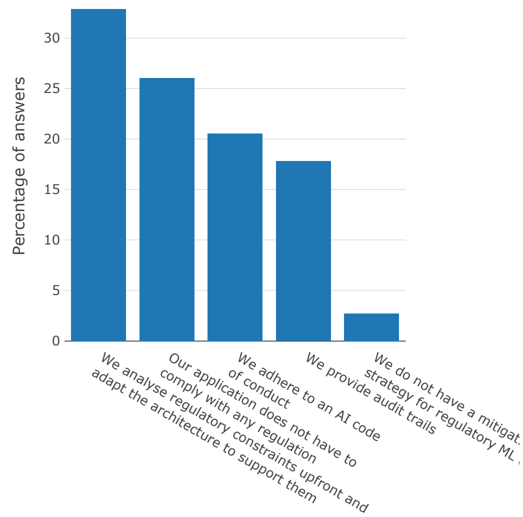


Figure 35: Survey solutions to practice 4.

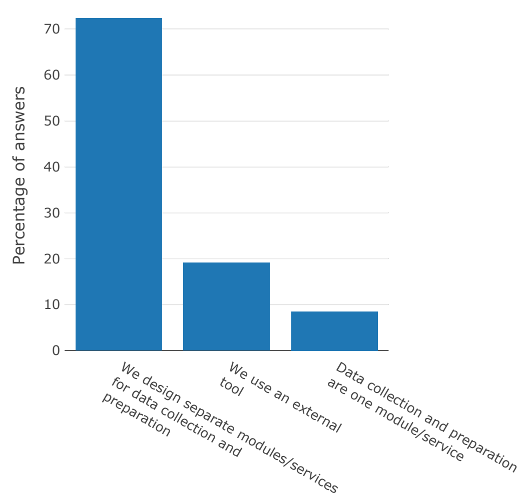


Figure 36: Survey solutions to practice 5.

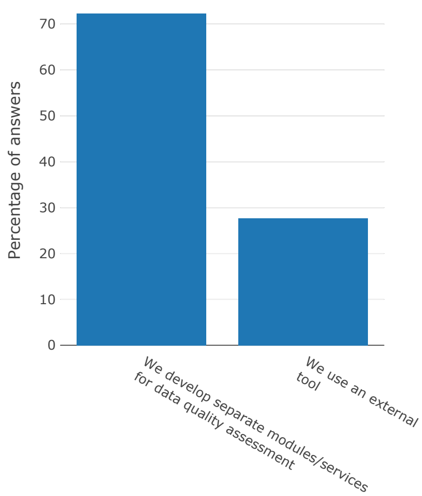


Figure 37: Survey solutions to practice 6.

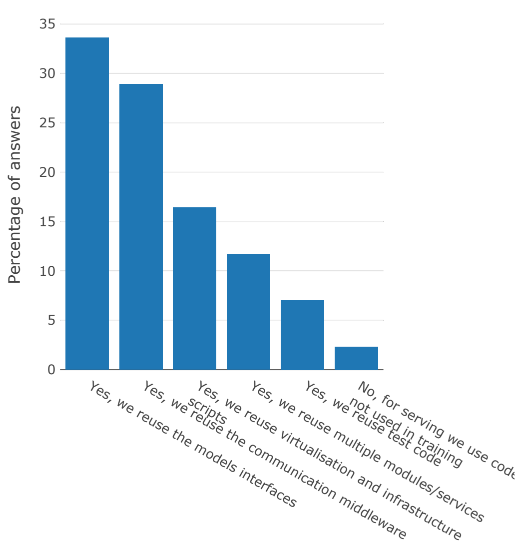


Figure 38: Survey solutions to practice 7.

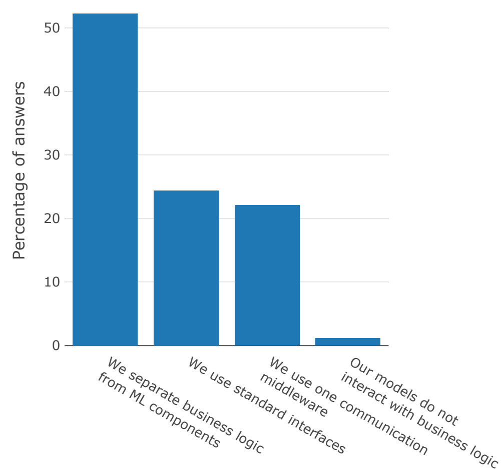


Figure 39: Survey solutions to practice 8.

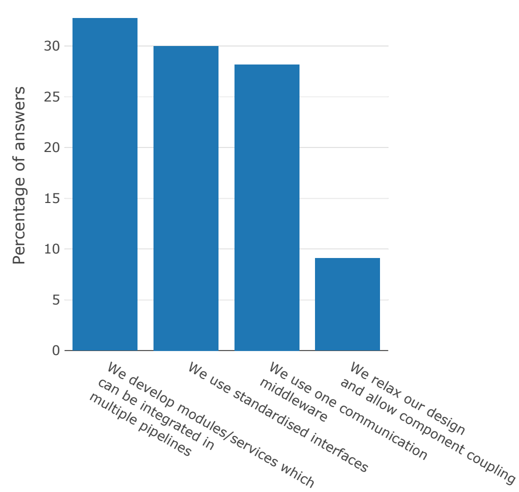


Figure 40: Survey solutions to practice 9.

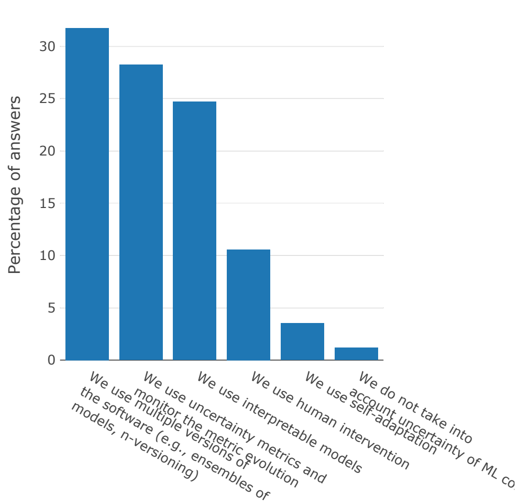


Figure 41: Survey solutions to practice 10.

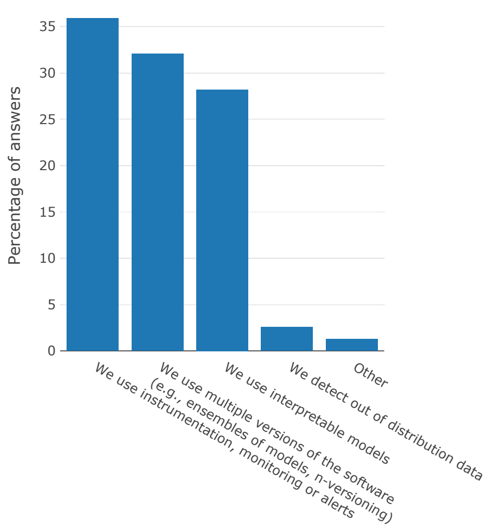


Figure 42: Survey solutions to practice 11.

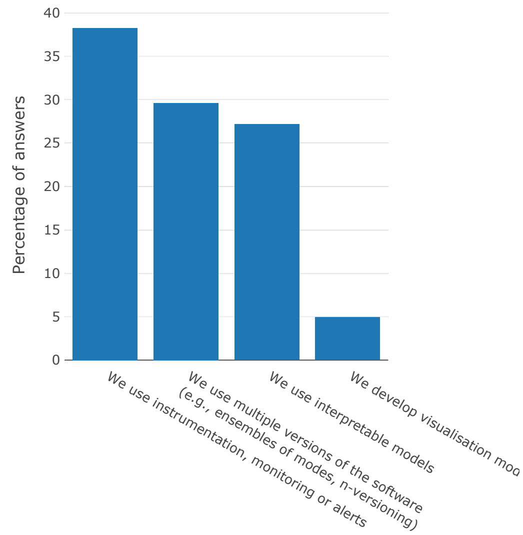


Figure 43: Survey solutions to practice 12.

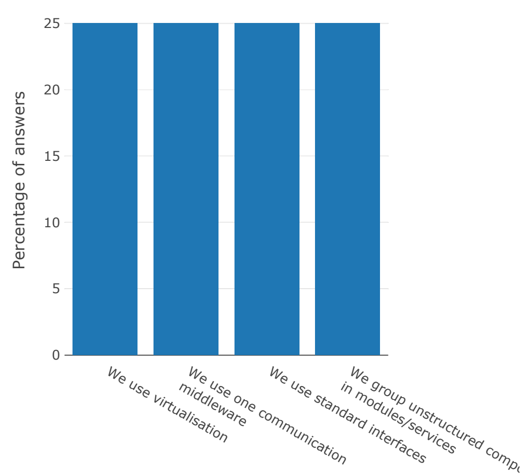


Figure 44: Survey solutions to practice 13.

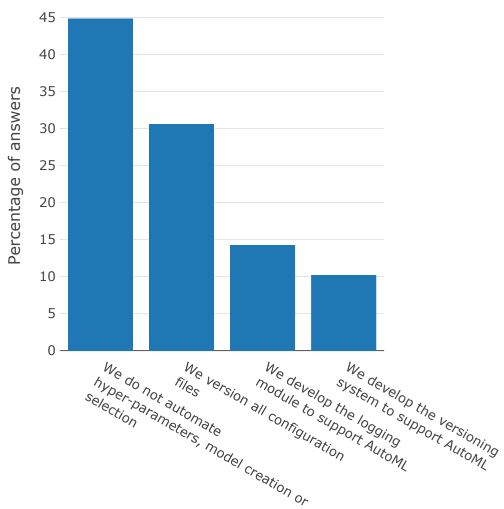


Figure 45: Survey solutions to practice 14.

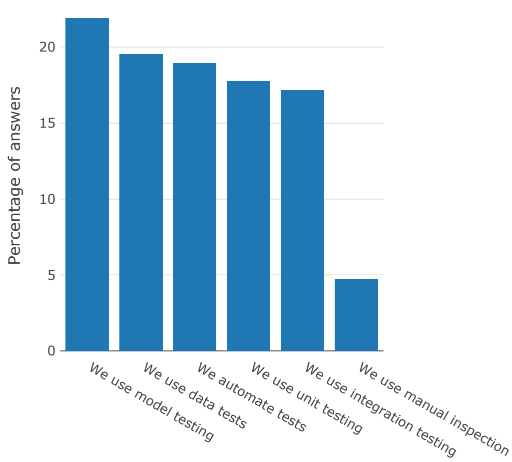


Figure 46: Survey solutions to practice 15.

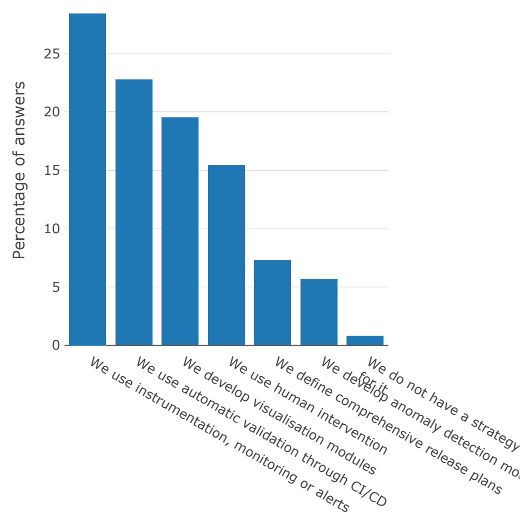


Figure 47: Survey solutions to practice 16.

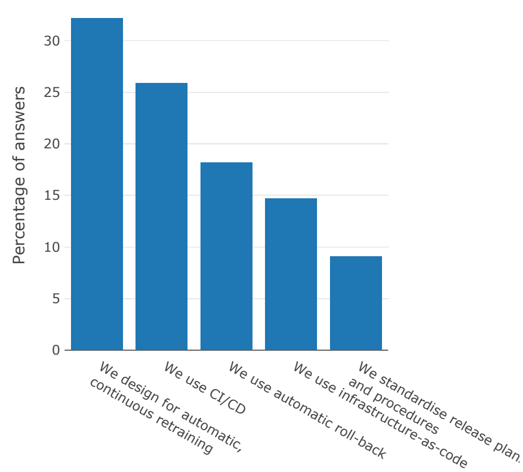


Figure 48: Survey solutions to practice 17.

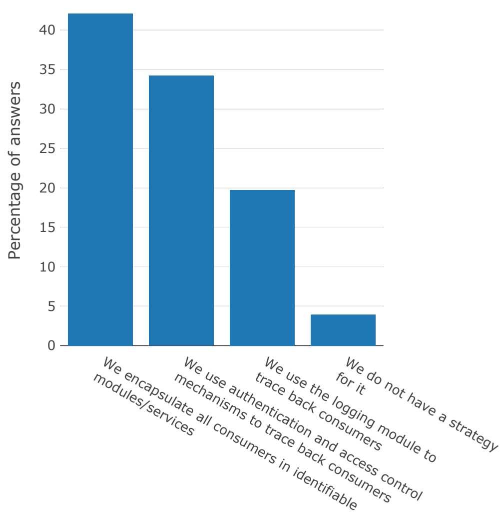


Figure 49: Survey solutions to practice 18.

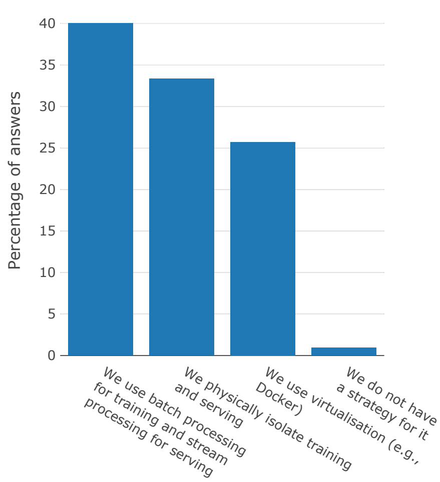


Figure 50: Survey solutions to practice 19.

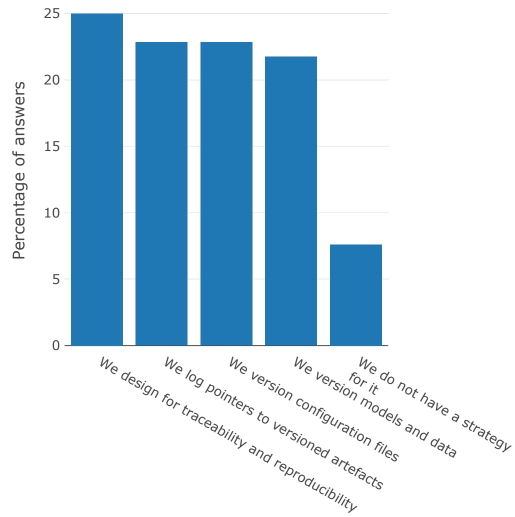


Figure 51: Survey solutions to practice 20.

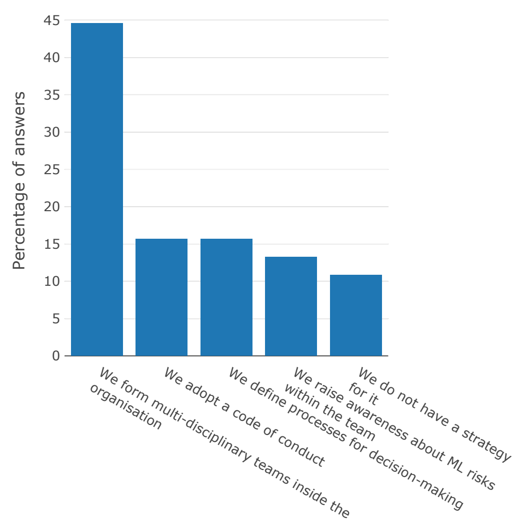


Figure 52: Survey solutions to instrumentation.

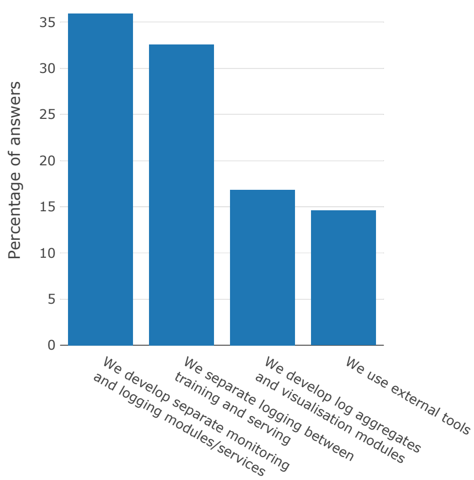
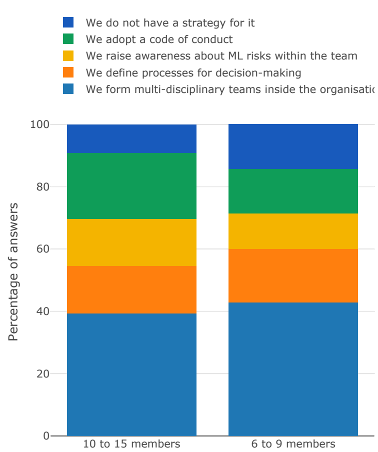


Figure 53: Survey solutions to interview challenges, grouped by team size.



J THEME DESCRIPTION

Run simulations to gather data. When designing systems with ML components, the information regarding data, the suitable ML techniques, or the cost of infrastructure is incomplete. Moreover, users have unrealistic expectations regarding the quality that ML can provide, in relation to the resources available. To get a better understanding of the project and detect issues as early as possible, it is recommended to run simulations before defining the SA. The simulations may include data flows, running ML models on open source data that resembles the project task, deploying and scaling mock models, etc.

Use past experience. Expert opinion and past experience can be used in designing systems with ML components. Nonetheless, past experience should be paired with context information, as distinct ML techniques have different inherent requirements.

Measure and document uncertainty sources. To better evaluate the resources available, and create accurate expectations regarding the quality of ML components, it is recommended to measure and document sources of uncertainty. For example, uncertainty regarding data set size, data quality, regulatory constraints, number of users, etc. In particular, the data set size and quality may lead to bottlenecks early in the pipeline.

Use metrics as functional requirements. Compared to traditional software, ML components lack clear functional requirements. To overcome this drawback, it is recommended to use concrete, measurable, metrics as functional requirements. Examples of metrics are, accuracy, F1-score, but also metrics for trustworthy ML such as robustness, bias, etc.

Include understandability and explainability of the outputs. When translating requirements to measurable metrics, it is recommended to include metrics for understandability and/or explainability of the output of ML components. These metrics help to analyse and understand the data and the ML components. Moreover, they help shape future decisions such as choosing ML techniques suitable for the task, fall-back mechanisms, n-versioning, etc.

Analyse regulatory constraints up-front. Although ML/AI specific regulatory constraints are, at the moment, in draft phase, their impact on systems with ML component is expected to grow in the near future. By analysing the regulatory constraints upfront, architects can prepare the system for compliance and audits. For example, regulatory constraints may enforce privacy requirements, which will translate to SA decisions for using privacy preserving ML techniques, privacy data management methods, etc.

Adopt an AI code of conduct. AI code of conducts help shape both a system and the organisation. For example, a code of conduct can stipulate the responsible use of ML/AI, which will translate to SA decisions for using interpretable models. Organisation wise, AI code of conducts will guide the decision processes.

Design audit trails. Audit trails help to analyse and understand a system, even without context knowledge. Since ML components are intrinsically opaque, it is recommended to think of audit traces when defining the SA. This process will help to define separate models to aggregate logs, provide visualisations, or automate reports and audit traces. Moreover, it helps third parties to assess compliance to regulatory constraints.

Design separate modules/services for data collection and data preparation. Data collection and preparation are experimental processes, which may result in a jungle of scrapes, joins and sampling steps. Therefore, the modularity and reuse of data modules is diminished. In order to avoid high coupling and facilitate code reuse, it is recommended to separate data collection and preparation into separate modules, which can be imported/deployed and orchestrated independently.

Integrate external tools for data collection and preparation. Multiple tools already provide data collection and preparation capabilities (e.g., Snorkel[1]). In case the project does not have tight data management constraints (e.g., privacy), architects can choose to integrate external tools. When choosing the tools for these task, architects must ensure they can be integrated with the system, and orchestrated in different services.

Design separate modules/services for data quality assessment. Data quality assessment consists of testing the data for missing data values, distribution skews, drifts, etc. In order to avoid coupling between data components, it is recommended to separate data quality assessment in individual modules/services. The modules should be independently imported/deployed and integrated in multiple pipelines, ensuring modularity and reusability.

Standardise model interfaces. To facilitate interoperability and reusability between training and serving, it is recommended to package ML models in standard interfaces. Choosing an interface type is project specific, and can include REST APIs, gRPCs, etc. Nonetheless, the models' interfaces should be the same as the interfaces for other components in the system.

Use one middleware. To enhance interoperability between ML and other components, and better integrate the ML experimental work-flow with traditional software, it is recommended to use a communication middleware between components. Even more, the middleware can be reused between training and serving, because the impact from training on infrastructure should not be high. For example, if the system uses a message queue in production, thee number of messages exchanged for training will be small.

Reuse virtualisation and infrastructure scripts. Infrastructure-as-code and virtualisation (e.g., Docker) should be adopted in all systems with ML components. Moreover, this code for infrastructure and virtualisation can be reused between training and serving, e.g., by holding states in configuration files.

Separate business logic from ML components. To distinguish errors between ML components and business logic, it is recommended to separate their concerns and development. The separation can be done by using independent modules/services for each concern. Moreover, standardisation of interfaces and the use of one middleware facilitate interoperation between business logic and ML components.

[1] https://www.snorkel.org/

Standardise interfaces. As mentioned in *standardise model interfaces*, standardisation of interfaces should be adopted between all components in a system.

Design independent modules/services for ML and data. ML components are coupled (and dependent) to data components. To minimise the coupling between the two, it is recommended to design independent modules or services for ML models and data (as also recommended in the data category).

Relax coupling heuristics between ML and data. Since ML and data components are highly coupled, there are use cases when the coupling can not be reduced. Therefore, for ML components it is recommended to relax coupling heuristics. Nonetheless, the heuristics should not be relaxed for other (business) logic.

Use n-versioning. Ensembles of ML models are used to decrease the risk of over-fitting, better approximate prediction uncertainty or facilitate interpretability. Thinking of ensembles as n-versioning helps to understand the role of each model, and its integration in the system. Moreover, it helps to separate the voting logic from the ML logic. An example is using an interpretable or rule based model as back-up for a black-box model.

Design and monitor uncertainty measures. Besides the metrics used as functional requirements (e.g., accuracy), it is recommended to monitor the uncertainty of ML components because it can signal degradation of models and silent failures.

Employ interpretable models. Interpretable models help to understand the decisions of ML components both by developers and users. Therefore, they should be the first choice for all systems with ML components. In case interpretable models can not satisfy functional requirements, they should be developed as secondary models in n-versioning.

Use human intervention. If possible, ML models should be assessed by team members (e.g., developers, data scientists, etc.).

Use metric monitoring and alerts to detect failures. ML components can fail silently, i.e., their predictions can be erroneous, in spite of the fact that they are running. To detect silent failures, it is recommended to use continuous monitoring of metrics, and implement alert systems which can notify incidents (e.g., decrease in accuracy, increase in uncertainty). When designing the alert modules, it is important to consider that alerts may be interpreted by team members that did not assisted the development. Therefore, alerts should be comprehensive enough to be interpreted by external actors.

Instrument the system to the fullest extent. Instrumentation is key to detection of failures, distribution shifts, or, more generally, malfunctions of ML components. Therefore, it is recommended to dedicate attention to define as many metric as possible, in order to analyse ML components. Instrumentation also facilitates incident management in production.

Design log modules to aggregate metrics. Log modules should be designed to allow fast mining of data. Nonetheless, in preparation for time constrained situations, such as incident management, log modules should be designed to already aggregate metrics and save high level conclusions. Aggregates are also helpful for audit reports.

Design log modules to visualise metrics. Besides metric aggregation, the design of visualisations modules to monitor ML components is recommended.

Wrap components in APIs/modules/services. Using generic packages for ML can result in large amounts of support code that connects ML components to other parts of the system. This glue code is costly, because it makes systems dependent on packages versions. To combat glue code, black-box packages can be wrapped in common APIs or independent modules/services.

Version configuration files. To reproduce previous ML experiments, and collect historical data about successful experiments, it is recommended to version configuration files among other artefacts, such as training and testing data, or models.

Design the log system to support AutoML. Data supporting the automation of ML development can be collected from logging or versioning. Designing the log system to support AutoML consists of designing modules for data storage and retrieval. An example is the creation of data sinks to support AutoML.

Design the version system to support AutoML. Similar to designing the logging system to support AutoML, the versioning system can support data collection and retrieval for AutoML. For example, by creating repositories dedicated to AutoML data.

Design model tests. Model testing are similar to unit tests, but for ML models. Model tests verify if models work with the data used in serving, and if their output satisfy predefined conditions. Besides implementing the tests, it is recommended to encapsulate them in independent modules, which can be orchestrated in different processes, if needed.

Design data tests. Data tests check the data satisfies predefined conditions; such as format, missing fields, distributions, etc. Similar to model tests, data tests should be encapsulated in independent modules, which can be reused in multiple pipelines. Moreover, it is desired to reuse data tests between training and pipelines.

Use integration tests. Integration tests verify the integration of ML components with other components in the system, and prevent cascading errors. Therefore, they are recommended for all systems with ML components.

Use CI/CD.. Automation in building, testing and deployment of systems with ML components facilitates fault isolation, increases reliability and team agility. Therefore, CI/CD is recommended for all projects with ML components.

Use unit tests. Besides model and data tests, it is recommended to write unit tests for all code; as in traditional software development.

Use data ownership for test modules. Consider individual data splits owned by test modules. In this way, one can ensure that test data is not mistakenly used for training. Moreover, test modules orchestrated in different pipelines will rely on independent data samples. However, it is recommended to ensure the data is continuously refreshed.

Use metrics and CI/CD for validation. In order to validate models for production, it is recommended to design a metric suite, and automate its verification through CI/CD.

Use alerts for validation. In case a model is promoted/discarded for production, it is recommended to announce it to the team through an alert module.

Use human intervention for validation. In order to maintain human oversight, it is recommended that team members analyse models promoted to production.

Design for automatic continuous retraining. Maintenance an evolution of ML components requires continuous retraining. In is recommended to design the system s.t. this process can be automated. For example, by developing modules that can deploy infrastructure and orchestrate data and training modules.

Use automatic rollback. If, due to changes in the input data or undetected skew, a deployed model performs sub-optimal, it should be rolled back to an earlier, better performing version. Designing a process for automatic roll-back minimizes the time a deployed model with sub-optimal performance is kept in production.

Use infrastructure-as-code. Setting up the infrastructure for ML components (e.g., starting new machines, transferring the data) is a tedious process. In order to increase the agility, it is recommended to automate this task.

Adopt standard release processes. To avoid human errors and sub-optimal models from being deployed, it is recommended to design and adopt standard releases processes. Since ML teams are heterogeneous, some team members may be unaware of the processes for model release or roll-back. Shared within the team, standard processes empower members to act when faced with incidents.

Encapsulate ML components in identifiable modules/services. To manage the dependencies and consumers of ML components, it is recommended to encapsulate them in identifiable modules/services. The unique identifiers can be used to trace back consumers, and should be used to log all interactions with ML components.

Use authentication and access control. In order to forbid access to ML consumers, it is recommended to implement authentication and access control policies. Similar policies should be implement for the data used by ML components.

Log consumers of ML consumers. Unique identifiers of ML components can be used to trace back consumers, and should be used to log for all interactions with ML components.

Design for batch processing for training and stream processing for serving. To balance latency, throughput, and fault-tolerance, needed for training and serving it is recommended to adopt a lambda architecture, i.e., use batch processing for training and stream processing for serving.

Physically isolate the workloads. To scale training and serving workloads, it is recommended to physically isolate them, i.e., deploy on distinct physical hardware. This is automatically done when using cloud services.

Use virtualisation. To abstract from the software suite needed for ML, and increase reproducibility and agility, it is recommended to use virtualisation (e.g., Docker).

Design for traceability and reproducibility. In a system with ML components, traceability and reproducibility are inherent requirements which need to be satisfied. Different strategies can be applied, within which two methods are common: logging pointers to versioned artefacts, and versioning of data and model related artefacts.

Log pointers to versioned artefacts. Tracing decisions back to the input data and the model's version can be difficult. It is therefore recommended to log production predictions together with the model's version and input data.

Version configurations, models and data. To reproduce previous ML experiments, one needs more than just the executable code. Versioning the training and testing data, the final model, and all configuration files is complementary to versioning the code.

Form multi-disciplinary teams. ML components require the integration of different technology stacks responsible for infrastructure, data management, model training and serving. To tackle the challenges at different levels, it is recommended to form multi-disciplinary teams consisting of members responsible for different levels of the stack.

Define processes for decision making. In many cases, decisions have to be made under uncertainty and incomplete information. To enhance the ability of a team to make the right decisions at the right time, it is recommended to define and enforce a standard process across team members. ML adds more uncertainty to a system, and increases the importance of standard processes for decision making.

Raise awareness of ML risks within the team. Even when used with good intentions, ML can negatively impact on the society. Therefore, it is recommended to inform team members working at different levels of the stack about ML risks.