TU/e EINDHOVEN UNIVERSITY OF TECHNOLOGY

Arvisan

Citation for published version (APA):
Kakkenberg, R., Rukmono, S. A., Chaudron, M. R. V., Gerholt, W., Pinto, M., & de Oliveira, C. R. (2024). Arvisan: an Interactive Tool for Visualisation and Analysis of Low-Code Architecture Landscapes. In *MODELS Companion '24: Proceedings of the ACM/IEEE 27th International Conference on Model Driven Engineering Languages and Systems* (pp. 848-855). Association for Computing Machinery, Inc. https://doi.org/10.1145/3652620.3688331

Document license:
CC BY-NC-SA

DOI:
10.1145/3652620.3688331

Document status and date:
Published: 31/10/2024

Document Version:
Publisher's PDF, also known as Version of Record (includes final page, issue and volume numbers)

Please check the document version of this publication:

• A submitted manuscript is the version of the article upon submission and before peer-review. There can be important differences between the submitted version and the official published version of record. People interested in the research are advised to contact the author for the final version of the publication, or visit the DOI to the publisher's website.
• The final author version and the galley proof are versions of the publication after peer review.
• The final published version features the final layout of the paper including the volume, issue and page numbers.

Link to publication

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

• Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
• You may not further distribute the material or use it for any profit-making activity or commercial gain
• You may freely distribute the URL identifying the publication in the public portal.

If the publication is distributed under the terms of Article 25fa of the Dutch Copyright Act, indicated by the "Taverne" license above, please follow below link for the End User Agreement:

www.tue.nl/taverne

Take down policy
If you believe that this document breaches copyright please contact us at:

openaccess@tue.nl

providing details and we will investigate your claim.

Download date: 28. Mar. 2025

Arvisan: an Interactive Tool for Visualisation and Analysis of Low-Code Architecture Landscapes

Roy Kakkenberg
Eindhoven University of Technology
Eindhoven, The Netherlands

Satrio Adi Rukmono
s.a.rukmono@tue.nl
Eindhoven University of Technology
Eindhoven, The Netherlands

Michel R. V. Chaudron
m.r.v.chaudron@tue.nl
Eindhoven University of Technology
Eindhoven, The Netherlands

Wim Gerholt
Royal Vopak
Rotterdam, The Netherlands

Miguel Pinto
Royal Vopak
Rotterdam, The Netherlands

Claudio Ribeiro de Oliveira
Royal Vopak
Rotterdam, The Netherlands

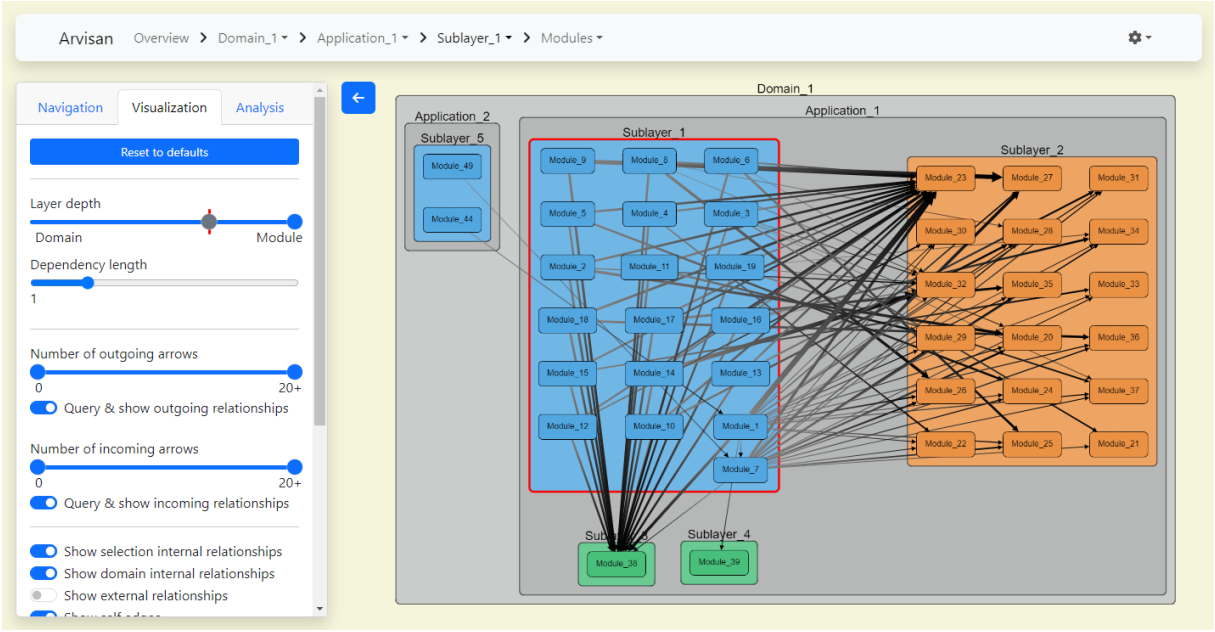

Figure 1: Arvisan visualisation of all dependencies of an OutSystems application's end-user sublayer.

Abstract

This paper introduces Arvisan, an interactive tool for visualization and analysis of low-code architecture landscapes. Arvisan improves upon existing low-code architecture assessment tools, particularly Discovery, which is used to verify the technical integrity and adherence to architectural rules in OutSystems applications. Arvisan offers overviews as a sliding window across abstraction levels, highly interactive visualization, and a variety of quality analyses. Arvisan uses a graph-based visualisation approach to architecture analysis. The information on the components of OutSystems is modelled as a labelled property graph. The tool is applied to an industrial-scale application landscape. Arvisan provides several advantages over Discovery: it offers a unified overview, efficient navigation, overview on different levels of abstraction, and the ability to detect indirect dependency cycles. The tool was evaluated with the OutSystems low-code platform and received positive feedback from developers. Arvisan contributes to the arsenal of tools for analysing architectures of low-code systems.

CCS Concepts

• **Software and its engineering** → **Maintaining software**; **Software maintenance tools**; **Layered systems**.

Keywords

low-code architecture analysis, low-code architecture visualisation, software architecture analysis, software architecture visualisation

This work is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike International 4.0 License.
MODELS Companion '24, September 22–27, 2024, Linz, Austria
© 2024 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0622-6/24/09
https://doi.org/10.1145/3652620.3688331

ACM Reference Format:
Roy Kakkenberg, Satrio Adi Rukmono, Michel R. V. Chaudron, Wim Gerholt, Miguel Pinto, and Claudio Ribeiro de Oliveira. 2024. Arvisan: an Interactive Tool for Visualisation and Analysis of Low-Code Architecture Landscapes. In *ACM/IEEE 27th International Conference on Model Driven Engineering Languages and Systems (MODELS Companion '24), September 22–27, 2024, Linz, Austria.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3652620.3688331

1 Introduction

As software increasingly becomes an integral part of business operations, companies want to automatise more processes to increase productivity, reduce costs, and improve customer experiences. An effective way to ensure the software used accommodates all current and future functionality needs is to develop your own software. In such a case, the company is the owner of its own automation processes, allowing full control of the implementation. However, developing large software systems is risky and potentially expensive, especially for companies whose core business is not software engineering.

Low-code platforms aim to solve this problem by simplifying the development process. By using visual programming, ready-to-use components, and automating the deployment process, software development becomes accessible to people with little to no software development knowledge. Even though low-code platforms bring licensing costs, our experts at Royal Vopak[1] have found that the accessibility and added productivity of developers often outweigh these costs. OutSystems, founded in 2001, is one such platform.

As a low-code system grows, like any other kind of software, its maintenance becomes more complicated. Discovery [15] is a tool available on the OutSystems Forge, the marketplace for reusable modules and applications for OutSystems applications. Discovery allows architectural analyses, which help the effectiveness of system maintenance. Its goal is to summarise the properties and hierarchical organisation of the low-code components on a platform server. It shows dependencies by listing components that are *producers* and components that are *consumers* of each producer. Most importantly, it lists violations of OutSystems' architectural guidelines as specified in the Architecture Canvas [5].

However, the capabilities of Discovery are mostly limited to listing architecture properties and issues. It does not effectively visualise components in the context of others, except for the type of layer they are in. This makes it difficult to get a general overview of the landscape and identify potential issues. Additionally, our experts mentioned that Discovery's performance is suboptimal. Users find it time-consuming to locate information about a specific application or module, as it takes a considerable time to load the required information. This intra-team observation is supported by some posts in the OutSystems forum discussing performance issues and lack of filtering capabilities in Discovery [12, 14].

In response to these challenges, we have developed Arvisan[2] (the Architecture Visualiser & Analyser), a visualisation-based architecture analysis tool for OutSystems. Arvisan uses data from the OutSystems metamodel to interactively visualise the components of a low-code software system. The visualisation data then serves as a foundation for further analysis. The goal of Arvisan is to independently assess the software architecture landscape and support software engineers in their daily work. This paper elaborates on how Arvisan achieves its visualisations and analyses and how these improvements compare to Discovery's.

2 Context & Related Work

This section describes the context of our contribution, i.e., the OutSystems low-code platform, and related work on architectural analysis on the platform.

2.1 The OutSystems low-code platform

OutSystems[3] is a leading [10] low-code platform designed for the development of mobile and web enterprise applications.

An OutSystems *application* consists of one or more *modules*. A module is a container for all the *elements* required to implement a part of an application, such as data models (or *entities*), user interface *screens*, and business *logic*. An *action* is a type of logic element that defines logic flows that run on the application. An action of certain types can be exposed to other elements as a *function*. As such, the hierarchy of *application→module→(logic) element* in the OutSystems platform maps to a typical hierarchy of *program→module→subroutine* in structural programming.

The OutSystems Architecture Canvas [5] dictates that an OutSystems module should belong to one of three *layers*. Modules in the topmost layer, *End-user*, provide functionality to the end user. The middle layer, *Core*, provides services around business concepts, while the bottom-most layer, *Foundation*, provides services to connect to external systems or to extend the framework on which the application is developed. These layers are further decomposed vertically into *sublayers*. A module of a certain (sub)layer should never depend on a module from layers above it. By definition, the respective layers above map to *Presentation*, *Domain Services*, and *Technical Services* layers in a typical layered architecture [16].

Figure 2 illustrates how components within an OutSystems application relate to each other.

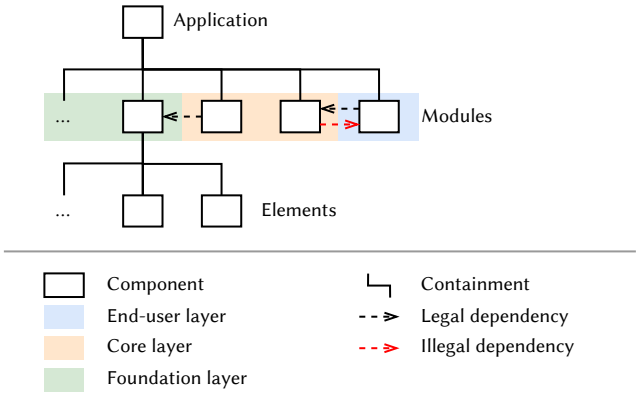

Figure 2: An illustration of components in an OutSystems application.

[1] https://www.vopak.com/
[2] https://github.com/Yoronex/arvisan-backend/tree/v0.10.1
[3] https://www.outsystems.com/

2.2 Related work

Discovery [15] is the current state of the art in OutSystems software architecture analysis. As mentioned in the introduction, it allows OutSystems architecture landscape analyses. To aid in organising OutSystems applications, Discovery introduces an additional level of hierarchy called *functional domain* (*domain* for short), which comprises applications with the same underlying business concepts. Then, it creates overviews of all applications in a domain or all modules in an application. Each component in the overview shows the number of dependencies and whether the component contains any violations. For each overview, the user can filter on the domain, application, architecture layer, and architecture sublayer. Furthermore, it mentions if there are any upward references or direct dependency cycles. Discovery shows an application or module's *incoming* dependencies (consumers) and *outgoing* dependencies (producers). When selecting an application, Discovery lists the dependencies on the application and module level, but not which module belongs to which application; i.e., it does not visualise the hierarchy.

The OutSystems AI Mentor (previously known as the Architecture Dashboard [11]) is a tool that computes and visualises the amount of technical debt in OutSystems applications over time and between applications. It achieves this through the use of tables and colouring schemes. For each application, the AI Mentor explains the technical debt score by listing the detected antipatterns and quantifying the associated technical debt. Additionally, the AI Mentor provides suggestions on how to address these antipatterns. In summary, the tool primarily focuses on assessing the quality of individual applications or modules, without considering their context (e.g. dependencies with other modules). Consequently, an architecture landscape analysis is absent from the AI Mentor. This gap in functionality explains the existence of Discovery on the OutSystems Forge, as the AI Mentor is considered inadequate for analysing large applications.

BonCode, an independent company, offers insights into software quality using automated tooling [3]. Their primary focus is on providing metrics-based software quality scores and ratings over time. These metrics can be visualised at three levels: domain (referred to as *project* in the context of BonCode), application, and module. The *solution score* (representing the overall quality of an application) comprises three components: *decomposition* (assessing the modularity of the application), *architecture* (evaluating compliance with the Architecture Canvas rules), and *volume* (considering the application's size and complexity).

BonCode, at the time of writing, is developing an architecture visualisation. It visualises the tree structure of the different components at various levels: *application* level, *module* level, and *module function* level. Additionally, it includes dependency arrows between different components. While this layout works well for small visualisations, it becomes too large and cluttered for large datasets. However, their visualisations offer more detailed information about the elements within each module.

Despite BonCode's more comprehensive analyses of the architecture landscape compared to Discovery, their solution remains proprietary, and the exact computation of scores is not fully disclosed. To date, we have not identified any other tools for OutSystems architecture landscape analysis or research that perform software visualisations and/or analyses on low-code software solutions.

Model slicing [8] is a concept of selecting parts of a model that is relevant to the current task at hand. While the original concept applies for UML class diagrams, particularly ones reverse-engineered from source code, it is applicable to any reasonably complex model. Some mechanisms we use for simplifying the low-code model for viewing is analogous to model slicing.

3 Arvisan

In this section, we discuss the various functionalities that Arvisan implements to achieve its visualisation-based analysis.

3.1 Data Model and Source of Information

Arvisan models the information on the components of OutSystems as a labelled property graph (LPG) [1]. We include the information described in Section 2.1, along with additional *domain* and *sublayer* information as defined by Discovery. The schema of our LPG can be seen in Figure 3.

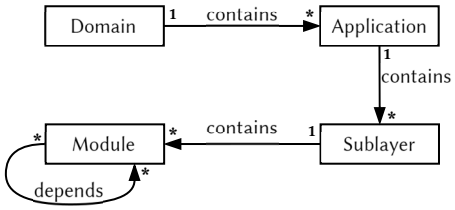

Figure 3: A graph schema for Arvisan's data model representing an ecosystem of OutSystems applications.

Containment Information. Arvisan considers the two highest levels of the OutSystems component hierarchy: *application* and *module*. We disregard *elements* in our analysis, as our focus is on architecture, and *elements* provide excessively detailed information. However, we add two more types of hierarchical information to our schema, directly inspired by the Discovery tool. The first addition is the *domain* nodes, which further organise *applications* into functional groupings. This is to reduce the complexity of the highest-level view of the application landscape in an organisation. The second addition is the inclusion of (*sub*)*layer* information as an intermediate level between *application* and *module*. This is helpful in simplifying dependency violation analyses. The application illustrated in Figure 2 is thus restructured as shown in Figure 4. Since OutSystems neither defines a *domain* nor explicitly maintains module-to-(sub)layer mapping, Arvisan relies on Discovery to define and provide such information.

Dependency Information. OutSystems tracks all dependencies between different modules in its internal databases. Specifically, it records which consumer modules (outgoing side of the dependency) are using which producer modules (incoming side of the dependency), and in which application these modules belong. Thus, an export from OutSystems' internal databases, combined with the information produced by Discovery as previously described, becomes the necessary input for Arvisan's visualisation and analyses.

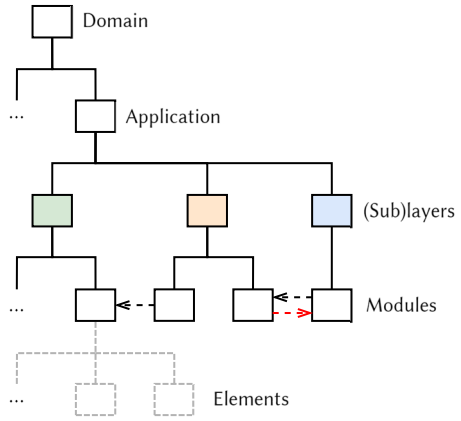

Figure 4: An application as illustrated in Figure 2, restructured according to Arvisan's hierarchical schema. (*Elements* are not used in Arvisan.)

3.2 Visualisation Approach

The graph structure created by Arvisan from the two inputs is then visually represented as a clustered graph [7]. In this representation, *modules* serve as nodes, *dependencies* between modules as edges, and *hierarchical containment* as clusters. Figure 5 illustrates the (partial) mapping from our graph schema to the visualisation.

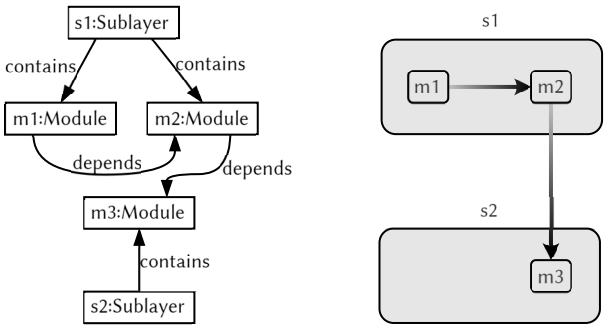

Figure 5: Illustration of how an instance of our graph model (left) translates to visual elements in Arvisan visualisation (right). Sublayers s_1 and s_2 become clusters that host the module nodes.

Arvisan's visualisation is interactive. It displays detailed properties of a component on demand and allows navigation by selecting a new starting point from the visible components, enabling users to interactively explore the architecture landscape.

The graph representing a complete software architecture landscape can become quite large. To ensure understandable visualisations and maintain tool performance, Arvisan employs two primary methods to reduce the size of the visualisation and its complexity.

First, Arvisan generates visualisations based on a user-selected starting point, which can be any node from the graph. It then queries dependency paths up to a user-defined maximum path length in a user-defined dependency direction (incoming and/or outgoing). Additionally, it retrieves information about the hierarchical context of the starting point and the incoming/outgoing dependants. This approach creates visualisations that focus on the context of the starting-point component while excluding other components that are not directly interacting with the currently selected component. In the visualisation, the starting point is always marked with a red border.

Second, to further reduce complexity when choosing a domain or application node as a starting point, we employ the *edge lifting* method to abstract the graph to higher levels. This method lifts dependencies from the module level to the level of the starting point. For example, when module M_1 of application sublayer S_1 depends on module M_2 of application sublayer S_2, this dependency is "lifted" into a dependency from S_1 to S_2. Duplicate edges created by the lifting are aggregated into a single dependency, resulting in a high-level view of the software architecture. This significantly reduces the complexity of the visualisation, as lower-level components do not need to be rendered. Effectively, this becomes a sliding window across abstraction levels – at one point in time, only a subset of the entire hierarchical levels is visible.

Note that edge lifting *adds* more information to the graph. In other words, the lifted edges do not replace existing edges at the more detailed level of abstraction. This allows Arvisan to display dependencies at either level of abstraction based on the user's preference.

3.3 Visualisation Filters

Arvisan provides filters to increase or decrease the size of the visualisation. These options allow users to tailor their visualisations to their specific needs. Some of these options are shown on the left side of Figure 1.

- Layer Depth: This slider implements the previously mentioned edge lifting. This thus defines the sliding window across abstraction levels.
- Dependency Length: Instead of only displaying direct dependencies, Arvisan can also visualise dependencies recursively. For instance, a dependency length of 2 would also render the dependencies of the selection's dependencies.
- Outgoing/incoming relationships: Arvisan can render either the incoming dependencies or the outgoing dependencies (or both) from the starting point. An *outgoing relationship* from the perspective of a selected component is an edge *from* that component to another component, and vice versa. Users can also limit the number of incoming/outgoing arrows per node.
- Selection Internal Relationships: These are dependencies fully contained within the selection (indicated by the red-bordered rectangle).
- Domain Internal Relationships: These are dependencies fully contained within the domain of the selection.
- External Relationships: These dependencies are not fully contained within the selection's domain. In other words, they either span completely different domains or have one end of the dependency arrow in a different domain.
- Type of Dependency: Although Arvisan does not maintain the details of elements within a module, the dependencies between modules include information about the element

type, e.g., whether module M_1 uses a logic, data structure, or user interface elements of module M_2. This dependency type can then be used for filtering visible edges.

3.4 Architectural Smells

Arvisan's visualisation highlights architectural smells, which arise from violations of architectural guidelines. OutSystems identifies two types of violations: cyclical dependencies and upward references (or sublayer violations) [6]. Architecture sublayers are ordered by their responsibilities to distinguish the level of reusability. For instance, an *End-user* sublayer should exclusively contain complete user interfaces, while *Foundation* sublayers define entities and integrations with other services. Components should not rely on components from a higher layer; for example, a *Foundation* module should not depend on an *End-user* module. During the visualisation building process, Arvisan detects these two violations and highlights them in the visualisation.

Cyclical dependencies are identified by examining cycles within the complete application landscape. Since the visualisation at any given point displays only a subset of all components, not all dependencies within a cycle may be visible. In such cases, Arvisan can still (partially) visualise the violation. When applying edge lifting, these cycles are also elevated to higher levels.

Upward dependencies are discerned in the visualisation. Arvisan includes blueprints for various sublayer violations, which are cross-referenced against the dependencies in the visualisation.

Arvisan represents these architectural violations in two ways: by colouring violating dependency arrows in red and by listing them in the left-side menu.

3.5 Metrics and Their Visualisation

In addition to the graph structure analyses described above, Arvisan provides metrics-based analyses. Metrics are computed within two different contexts: the visualisation context and the graph context.

Metrics calculated in the visualisation context are computed by the visualiser once a new graph is received from the graph builder. These metrics are context-dependent, as their utility is influenced by the visualisation settings that provide the necessary context. Other metrics are calculated based on the entire application context.

The following list describes the different metrics:

- Number of Incoming Dependencies: This interval metric is the total number of incoming function-level dependencies of all modules it contains. The metric is calculated in the visualisation context, allowing it to adapt to different visualisation settings.
- Number of Outgoing Dependencies: Similar to the previous metric, this interval metric is the total number of outgoing function-level dependencies of all modules it contains. Again, it is calculated in the visualisation context.
- Dependency Difference: To distinguish modules with predominantly incoming or outgoing arrows, the dependency difference metric has been introduced. This interval metric represents the difference between the number of incoming dependencies and the number of outgoing dependencies. It is available only as a colouring mode.
- File Size in Kilobytes: This interval metric reflects the component's file size in kilobytes. If unknown, it defaults to zero. Calculated during input data parsing, it requires the context of all children of an ancestor node and is therefore computed in the graph context.
- Number of Screens: Similar to file size, this interval metric counts the number of user interface screens in a module.
- Number of Entities: Like file size, this interval metric tallies the number of entity references.
- Dependency Profiles [4]: This categorical metric classifies modules based on their dependencies to other applications beyond their own. Calculated in the graph context, it considers potential exclusions from the visualisation that could lead to misclassification. To determine the dependency profile for each domain, application, or architecture sublayer, the dependency profiles of all its modules are summed. For individual modules, the dependency profile is stored as a vector representing their classification.
- Cohesion: This ratio metric aims to capture how interconnected all components within an ancestor node are. A high score indicates significant entanglement, while a low score suggests minimal entanglement. Similar to encapsulation scores, it is calculated in the graph context to avoid misinterpretations with varying visualisation parameters. The calculation involves dividing the number of inter-component module-level dependencies by the total number of components within the component, raised to the power of 1.5.

Visualising the metrics. Arvisan provides two methods to visualise metrics: 1) as colours for the graph nodes, and 2) as dimensions (width or height) of the graph nodes.

Colouring the graph based on a metric effectively compares metric values of different nodes. For example in Figure 6, the cohesion score clearly reflects the entanglement of modules within a sublayer, application, or domain. The graph is coloured based on the scale defined on the left side of the screen. Thus, we observe that the large yellow block has a high cohesion score, which seems reasonable given the numerous dependencies between the different modules in this block. All other blocks are purple, indicating a low cohesion score.

Another use case where colouring is beneficial, is in showing dependency profiles [4]. In Figure 7, modules in sublayer *#2* are coloured and arranged according to their dependency profiles. A module that depends on other sublayers (but is not used by other sublayers) is coloured blue (bottom right). Modules used by other sublayers (but not dependent on other sublayers) are coloured green (top left). Modules that both depend on and are used by other sublayers are coloured yellow (top right). Modules that do not interact with other sublayers are coloured red (bottom left). With this configuration, one can obtain a general understanding of module behaviours even without displaying dependency edges.

On the use of metrics to dictate a node's dimensions, one example where it is useful is to show whether a component functions more as a producer or a consumer. In Figure 8, the height of the nodes is set to correspond to the number of *outgoing* dependencies, while the width is set to correspond to *incoming* dependencies. With this

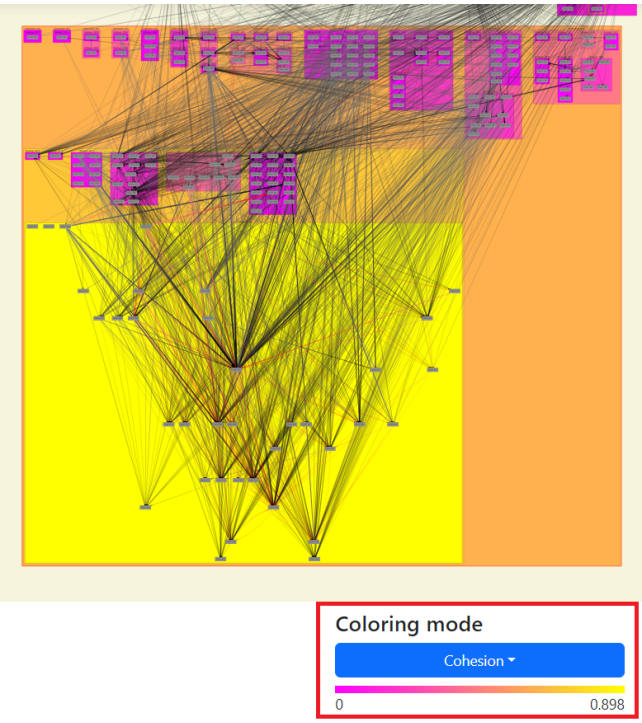

Figure 6: Arvisan visualisation with nodes coloured based on their cohesion score.

configuration, consumers-type of components appear as tall rectangles (more outgoing than incoming dependencies), while producers appear as wide rectangles (more incoming than outgoing). This allows for a quick mental matching – *end-user* modules should be tall, i.e., consuming *core* and/or *foundation* modules, while *foundation* modules should be wide.

4 Improvements over Discovery

The graph-based visualisation approach to architecture analysis offers several advantages over Discovery's list-based approach. As previously mentioned, Arvisan does not aim to replace Discovery; instead, it complements it. In certain cases, experts still prefer simple lists of dependencies and violations over visual representations. For instance, a quick check to determine whether an application satisfies the Architecture Canvas is better served by a straightforward list. Additionally, Arvisan leverages structured data from Discovery to create visual representations of Discovery's dataset.

In this section, we discuss some of the benefits that Arvisan provides compared to Discovery.

4.1 Unified overview

As an OutSystems application, Discovery can only access the metamodel of the Platform Server on which it is installed. However, large organisations may run multiple Platform Servers, each responsible for a different set of applications. Consequently, Discovery cannot combine metamodels to provide a single, comprehensive analysis of the entire landscape.

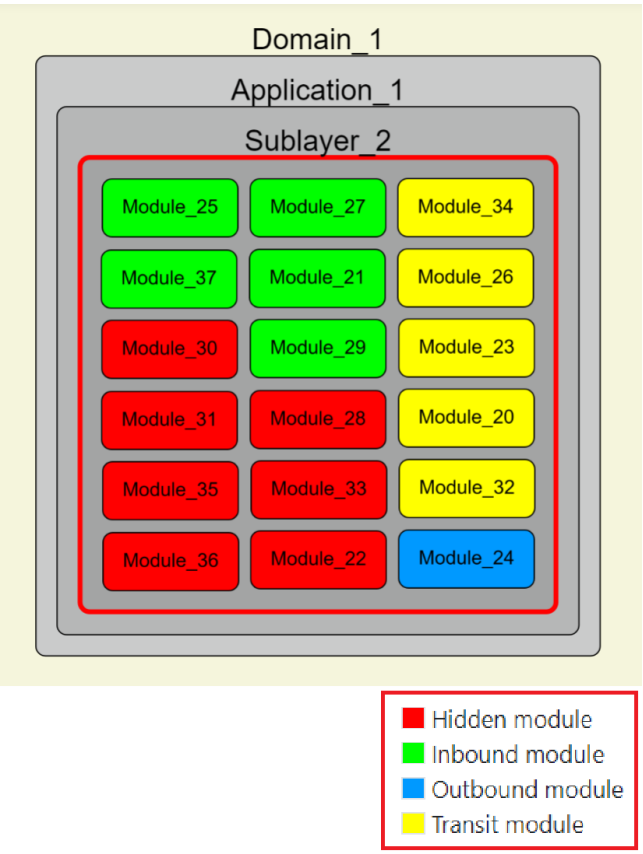

Figure 7: Arvisan visualisation with nodes coloured based on their dependency profiles.

Arvisan does not face this limitation. It combines multiple datasets from various OutSystems Platform Servers into a unified overview. This approach simplifies the process of locating specific domains, applications, or modules, as users no longer need to identify the server they are running on first.

4.2 Efficient navigation

Despite Arvisan's larger dataset (including more information), its initial loading is significantly quicker and more responsive than Discovery's. For instance, when searching for a specific module in Arvisan, users can simply query the entire database by name. Upon selecting the module, its dependencies immediately become visible.

Moreover, users can dynamically change the starting point of the visualisation through direct interaction. A mouse click on a node designates it as the new starting point, resulting in a fresh visualisation. This, combined with Arvisan's high performance in creating visualisations, enables efficient exploratory architectural analyses.

In contrast, Discovery requires loading all modules (a time-consuming process) before searching within the list for the desired module. Experts have noted the difference in wait time and the number of clicks, favouring Arvisan.

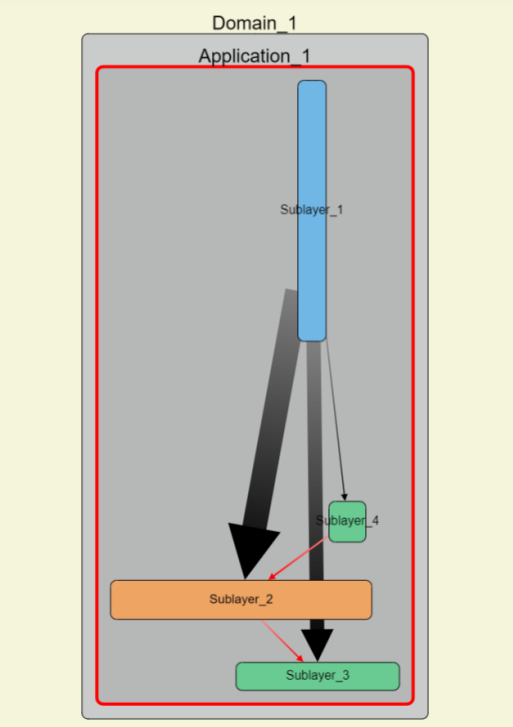

Figure 8: Using metrics to define node dimensions in Arvisan: number of outgoing dependencies as node height and incoming dependencies as width.

4.3 Overview on different levels of abstraction

Edge lifting not only drastically reduces the complexity of resulting graphs, but also makes it possible to analyse relationships between different elements on higher levels of abstraction. For example, Discovery cannot easily answer the question, "What applications have a dependency on domain *X*?" However, Arvisan can answer this question through the sliding window of abstraction. Figure 9 shows an example.

In addition to higher-level visualisations, the lower levels also provide easy overviews of the dependencies of single modules (see Figure 10). When the user chooses a module as the visualisation's starting point, Arvisan will only show the incoming/outgoing dependencies of that particular module. This feature is, for example, useful when a module is deprecated and all its dependencies need to be removed or replaced, or when a system module receives an update with breaking changes and the usage of these changes needs to be identified.

4.4 Indirect dependency cycles

Discovery can detect dependency cycles, but only direct cyclic references, where two modules depend on each other. Discovery cannot detect indirect cycles, where a cycle in the dependency graph has a length of at least three.

Arvisan can detect such larger cycles, as it creates a list of modules present in the visualisation that lie on at least one dependency cycle path. Furthermore, these violations are rendered in red in the visualisation, making them immediately visible to the end user. This approach allows large dependency cycles to become visible in an analysis, even if the cycle is not completely rendered on screen.

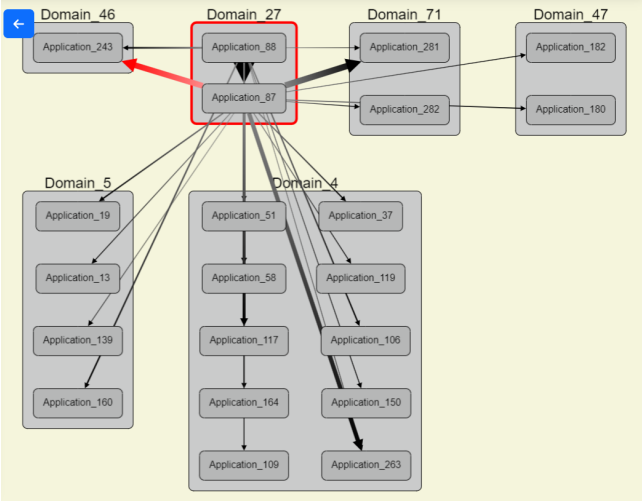

Figure 9: Arvisan visualisation of all outgoing application dependencies of a selected domain, *#27*, highlighting the existence of architectural smell(s) between applications *#87* and *#243* via red arrow.

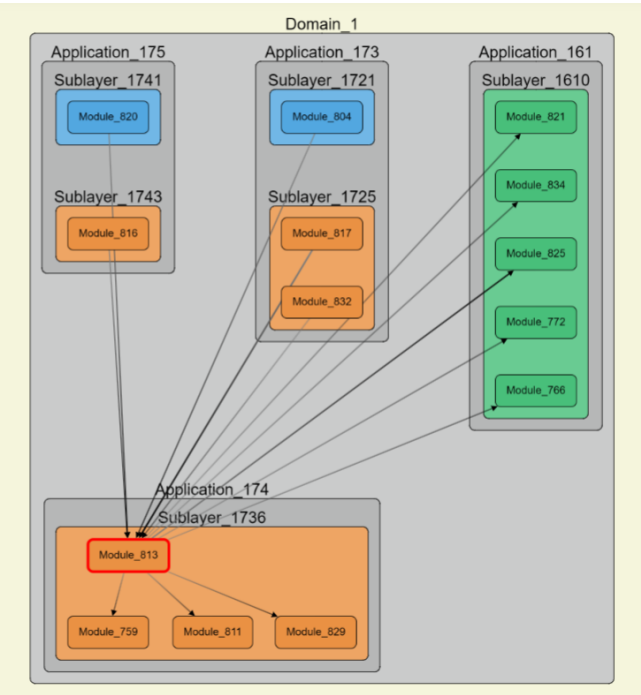

Figure 10: Arvisan visualisation of all incoming and outgoing dependencies of a selected module (*#813*) within a domain. Incoming dependencies are denoted by arrows going into module *#813*, and outgoing by arrows going from it.

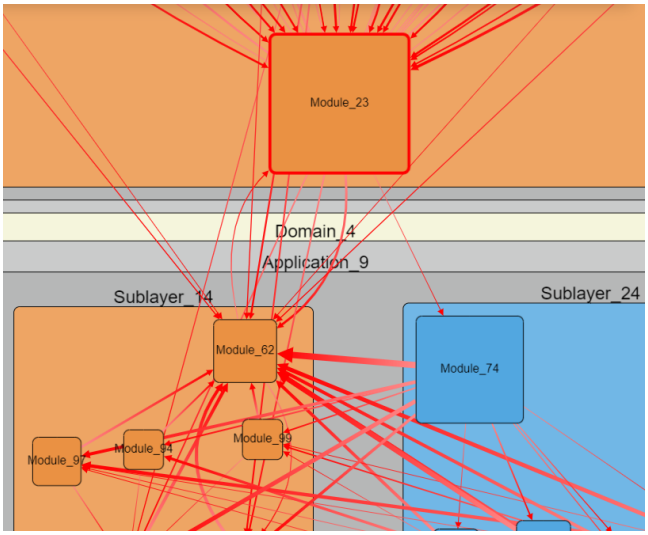

Figure 11: Arvisan visualisation of dependency cycles between two domains, e.g., the path from module *#23* to *#74*, *#62*, and back to *#23*.

5 Conclusion & future work

Arvisan is an architecture visualisation and analysis tool tailored for OutSystems software landscapes. Arvisan improves upon existing low-code architecture assessment tools, particularly Discovery, by: 1) offering complete overviews at different system levels, 2) providing highly interactive options for filtering and tailoring, which are valuable for managing large amounts of data, and 3) supporting a wide variety of analyses based on different types of software metrics. Specifically, the tool enhances the detection of dependency cycles.

The tool has been applied to an industrial-scale application landscape developed over the past 10 years and supported by more than 100 low-code software developers. We conducted interviews with these developers about the usefulness of the Arvisan tool, and the feedback was mostly positive. As a part of our contribution, we provide an anonymised subset of the evaluated application landscape dataset that can be imported in Arvisan [9].

Future work. This research largely built on existing layout algorithm for the visualisation. Future research could focus on improving the fit of the layout algorithm with the low-code paradigm. Furthermore, Arvisan could be extended by incorporating a time dimension in the visualisation, allowing for comparisons between different versions of the software. Finally, Arvisan was evaluated only with the OutSystems low-code platform, but many more platforms exist. It would be interesting to extend Arvisan to other platforms. In fact, since the concept of hierarchical structure and dependencies exists in software engineering in general, it would also be interesting to explore if Arvisan can aid in comprehending the structure of non-low-code systems. This should be possible by adapting source code graph representations [2, 13] into Arvisan's schema.

References

[1] Renzo Angles. 2018. The Property Graph Database Model. In *Proceedings of the 12th Alberto Mendelzon International Workshop on Foundations of Data Management, Cali, Colombia, May 21-25, 2018 (CEUR Workshop Proceedings, Vol. 2100)*, Dan Olteanu and Barbara Poblete (Eds.). CEUR-WS.org. https://ceur-ws.org/Vol-2100/paper26.pdf

[2] Mattia Atzeni and Maurizio Atzori. 2017. CodeOntology: RDF-ization of source code. In *The Semantic Web–ISWC 2017: 16th International Semantic Web Conference, Vienna, Austria, October 21-25, 2017, Proceedings, Part II 16*. Springer, 20–28.

[3] BonCode. 2024. BonCode for OutSystems – BonCode. Retrieved 17-05-2024 from https://boncode.nl/outsystems/

[4] Eric Bouwers, Arie van Deursen, and Joost Visser. 2011. Dependency profiles for software architecture evaluations. In *2011 27th IEEE International Conference on Software Maintenance (ICSM)*. IEEE, 540–543.

[5] OutSystems Community. 2023. The Architecture Canvas – OutSystems 11 Documentation. Retrieved 28-10-2023 from https://success.outsystems.com/documentation/best_practices/architecture/designing_the_architecture_of_your_outsystems_applications/the_architecture_canvas/

[6] OutSystems Community. 2024. Validating Your Application Architecture – OutSystems Best Practices. Retrieved 14-05-2024 from https://success.outsystems.com/documentation/best_practices/architecture/designing_the_architecture_of_your_outsystems_applications/validating_your_application_architecture/

[7] Qingwen Feng. 1997. *Algorithms for drawing clustered graphs*. Ph. D. Dissertation. University of Newcastle.

[8] Huzefa Kagdi, Jonathan I Maletic, and Andrew Sutton. 2005. Context-free slicing of UML class models. In *21st IEEE International Conference on Software Maintenance (ICSM'05)*. IEEE, 635–638.

[9] Roy Kakkenberg, Satrio Adi Rukmono, and Michel Chaudron. 2024. *Anonymized dataset of an OutSystems application landscape to be used in Arvisan.* https://doi.org/10.5281/zenodo.12645566

[10] Oleksandr Matvitskyy, Kimihiko Iijima, Mike West, Kyle Davis, Akash Jain, and Paul Vincent. 2023. Gartner Magic Quadrant for Enterprise Low-Code Application Platforms. https://www.gartner.com/en/documents/4843031

[11] OutSystems. 2022. Architecture Dashboard Is Now AI Mentor Studio | OutSystems. Retrieved 26-06-2024 from https://www.outsystems.com/product-updates/ai-mentor-studio/

[12] Joachim R. 2023. Discovery: Import Domains from LifeTime Issues. https://www.outsystems.com/forums/discussion/90864/discovery-import-domains-from-lifetime-issues/. Accessed: 2024-08-13.

[13] Satrio Adi Rukmono and Michel RV Chaudron. 2023. Enabling analysis and reasoning on software systems through knowledge graph representation. In *2023 IEEE/ACM 20th International Conference on Mining Software Repositories (MSR)*. IEEE, 120–124.

[14] Rajendra S. 2017. How to use Discovery tool? https://www.outsystems.com/forums/discussion/26315/how-to-use-discovery-tool/. Accessed: 2024-08-13.

[15] Architecture Team. 2023. *Discovery.* https://www.outsystems.com/forge/component-overview/409/discovery

[16] Rebecca Wirfs-Brock, Alan McKean, Ivar Jacobson, and John Vlissides. 2002. *Object Design: Roles, Responsibilities, and Collaborations*. Pearson Education.