NZa
NZa // ProMeSZ

YieldDD Tactical Assessment

yieldDD
software due diligence;

II Table of Contents

I Version history 2
II Table of Contents 3
III Management Summary 4
1 Introduction 6
1.1 Background 6
1.2 Research Objectives 6
1.3 Scope 6
1.4 Approach 6
1.5 Sources 7
1.6 Disclaimer 7
2 Software Solution 9
2.1 Introduction 9
2.2 Context 9
2.3 Architecture 9
2.4 History, roadmap, and backlog 11
2.5 Technical Debt and Bugs 11
2.6 Finding 12
3 Development 13
3.1 Introduction 13
3.2 Process 13
3.3 Team 13
3.4 Plan 13
3.5 Build 14
3.6 Test 14
3.7 Documentation and knowledge preservation 14
3.8 Third-party dependencies 14
4 Operation 16
4.1 Introduction 16
4.2 Process 16
4.3 Team 16
4.4 Deployment 16
4.5 Monitoring 17
4.6 Back-up and recovery 17
5 Code Quality 19
5.1 Introduction 19
5.2 Characteristics 19
5.3 Overall 20
5.4 Structure 20
5.5 Code 21
5.6 Security 21
5.7 Documentation 22
5.8 Testability 22
5.9 Database and T/Sql 23
6 Software Health 25
6.1 Introduction 25
6.2 Front end 25
6.3 Back end 25
6.4 Software Resiliency 26
6.5 Software Agility 26
6.6 Software Elegance 26

III Management Summary

- There is a lot of knowledge that only exists in the heads of the developers. We recommend that an effort is made to make all that knowledge explicit in documentation.
- There is no documentation regarding user flows in the application. This makes it hard to test these user flows consistently. We recommend to write test plans so that testers can do better work and so that a test automation engineer may write fully automated tests.
- A large part of the application development is done by writing SQL scripts for data updates during migrations, and to define stored procedures used by the applications. If the application is to be developed by another party, make sure they know and understand advanced SQL as part of the development process too.
- Access control to the application is managed by only allowing access to both server and client through the Citrix environment. We recommend reviewing the security of the server APIs to allow the back end of the application to be hosted outside of the local network.
- Existing back-ups have never been tested. We recommend to create a policy where those back-ups are tested on a regular basis to make sure that the back-ups work.

Introduction

1 Introduction

1.1 Background

Nederlandse Zorgautoriteit (NZa) aims to make care available and affordable. Everyone in the Netherlands is entitled to good care. A crown at the dentist. Care for someone who can no longer live at home because of Alzheimer's. Or treating a sports injury in the emergency room. The Dutch market is a very complex interplay between healthcare providers, insurers, consumers, politics, and a diverse field of strengths. NZa is in the middle of that field of forces.

NZa has asked YieldDD to perform a YieldDD tactical assessment (YTA) on Promesz.

1.2 Research Objectives

This assessment focusses on the following questions:

- What does the software solution entail? This covers the application context, architecture, maturity, and backlog of the solution.
- How is the software solution developed? This covers the team composition and development process.
- How is the software solution operated? This covers the deployment, operation, and monitoring of the solution.

In addition to these general research objectives, the assessment provides information to the following subjects:

- What is the software quality of the solution? This qualitative assessment covers the structure, code, testability, and documentation.
- What is the software health of the solution? This quantitative assessment covers the resiliency, agility, and elegance of the source code.

1.3 Scope

The scope of this assessment is the Promesz solution.

To answer the research questions, information has been acquired through questionnaires and interviews, a review of the documentation provided, and an assessment of the source code provided by NZa.

1.4 Approach

A basic understanding of the background and context of the software solution is targeted in surveys. These surveys provide input for an in-depth interview with the product owner and focusses on the various aspects of the application and software development process.

The solution context, architecture, vision and roadmap are summarized in chapter 2. In chapter 3 the development process and team composition covered, followed by the operation process and team in chapter 4.

To gain insight in the maintainability of the implementation, the source code of the inhouse-developed application was examined using static code analysis and Expert-Eye investigation. The software quality is covered in chapter 5 and the software health in chapter 6.

1.5 Sources

This assessment is based on the following sources:

Surveys:

- YieldDD General IT survey, 29/01/2022
- YieldDD Development survey, 29/01/2022

Documentation:

- Datamodel ProMeSZ.docx
  28/12/2022
- Hoofdlijnen architectuur Promesz.docx
  16/11/2022
- Technische en functionele beschrijving Promesz.docx
  28/12/2022

Source code:

- ProMeSZ.Client.20221227_1331.a4e20520.zip
  28/12/2022
- ProMeSZ.Server.20221227_1332.a4e20520.zip
  28/12/2022

The mentioned documents are available separately.

1.6 Disclaimer

This report is limited to the information made available to YieldDD by or via NZa. YieldDD trusts that everything provided to YieldDD by NZa is correct and complete. YieldDD cannot guarantee that its investigation and findings are correct and /or complete, nor that all the risks associated with the NZa applications are identified and inventoried. This report and supporting documents are exclusively intended for internal use by NZa and for the intended purpose and may not be published in whole or in part or provided to third parties without the explicit prior written consent of YieldDD. YieldDD is not liable for claims or damage of third parties as a result of publication of the report to third parties or as a result of possible inaccuracies.

Software solution

2 Software Solution

2.1 Introduction

This section provides insight in the system context of the application and the application architecture. The history of the application is summarized, and the roadmap, backlog and technical debt are addressed.

2.2 Context

Promesz is the application that is responsible for the distribution of funds for the hospital care sector, the so-called specialist medical care, which involves 20 billion euros annually.

The current system for distribution is called the DBC (Diagnose Behandelcombinatie) system. The DBC system is a combination of regulations and tables that are processed in ICT systems at hospitals, health insurers and central service parties. The system is not static, but changes at least twice a year via so-called DBC releases. Change requests are submitted, assessed and, after approval, processed in the regulations and tables. Hospitals record the treatments they perform and use the DBC system to determine which health care services can be declared.

The core output from Promesz are the decision trees which will be used to determine the distribution of funds.

2.3 Architecture

2.3.1 System technology

The solution consists of a server side and a client side part. The application is written in C# .NET Framework using WCF for the services, WPF with DevExpress components for the user interface, and Entity Framework for database storage using SQL Server.

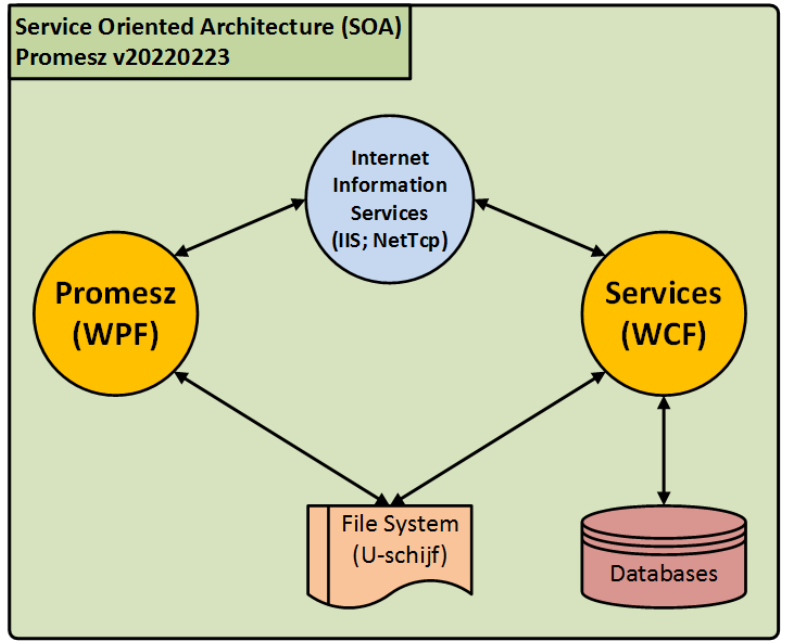


Figure 1: High level application architecture

2.3.2 Software architecture

The architecture is based on a clear vision by the lead software engineer on how to structure the application, and is based on Service Oriented Architecture (SOA). The client side is based on the Model-View-ViewModel (MVVM) principle.

The project is split into two C# solutions. The namespaces and responsibilities are clearly defined and documented.

Data can be exported to local file storage. Data can be imported using Excel. There are plans to replace the Excel import with native functionality in Promesz, where the data is managed from the user interface of Promesz instead of being imported.

2.3.3 Software landscape

Promesz connects to Formdesk to read change requests regarding the decision trees. These change requests are handled in Promesz as well, and all data regarding the requests are stored by and maintained through the application.

The decision trees are published on the NZa website and sent to VECOZO, an organisation which maintains connections to insurance companies and hospitals. VECOZO is not part of NZa, but fully dependent on the decision trees that NZa outputs. In theory there could be multiple management organisations like VECOZO, but in practice it is the only one. This may change in the future.

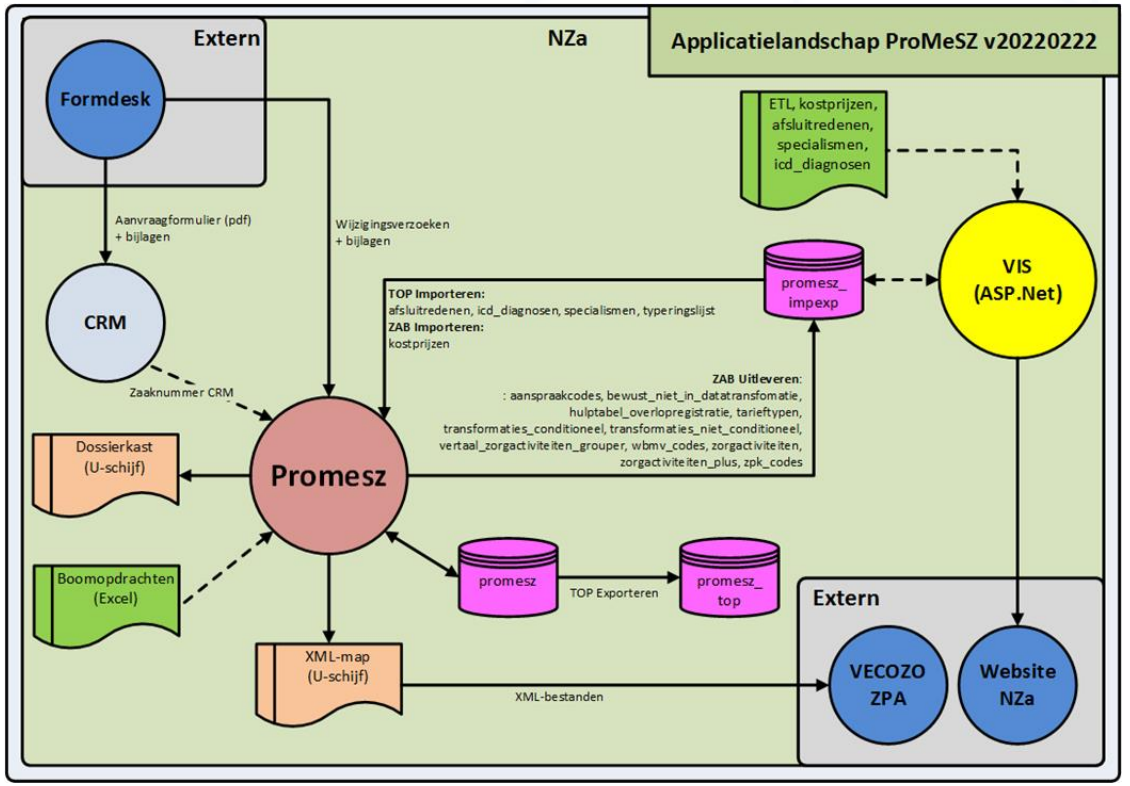


Figure 2: Promesz application landscape

2.4 History, roadmap, and backlog

Promesz is under active development. The project was greenlit in 2015, the first plans for development were made in 2016, and actual development was started in 2018. The program has been in use since 2020.

New features are added to the backlog. They may be picked up at the start of new sprints, priorities are set by stakeholders.

The reported number of all-time features listed on the backlog is represented in the table below.

Priority	# Features
Must have	1200
Should have	143
Could have	25
Would have	51

Table 1: All-time distribution of backlog items

The reported distribution of time spent on development is listed in the table below.

Product backlog	Backlog % of time spent
Features	80 %
Bugs	10 %
Architectural spikes	5 %
Technical debt	5 %

Table 2: Time spent on backlog items

2.5 Technical Debt and Bugs

The number of open issues with respect to technical debt is 37. Most technical debt is in a module called TOP. According to the development team there is no capacity or urgency to handle the technical debt. Technical debt is mostly related to the development of new feature requests, or related to bugs in the application. Technical debt is picked up when it relates to a new feature that will be implemented.

This table lists the number of bugs that have not been resolved.

Severity	#Known bugs
Critical	0
Major	12
Minor	38
Trivial	0

Table 3: Distribution of bug severity of open backlog items

New bugs are usually encountered when users are working with the application. The amount of bugs that are found coincides with the release of new decision trees because that is when most users use the application. This results in two annual peaks in the amount of new bugs.

Development

3 Development

3.1 Introduction

This section provides insight in the development process and team composition. Furthermore, feature planning, building, testing and documentation are addressed according to the stages defined in the Development cycle of the DevOps Lifecycle.

3.2 Process

The development team uses Scrum with 2-week sprints. There is no formal definition of done, tickets are closed after passing acceptance tests. There is no definition of ready.

The code is hosted on GitLab. Developers work on their own feature branch and merge the code to the development branch after they have finished. They do not use pull requests and by extension they do not do code reviews.

The development branch is used for application releases to the test and acceptance environments. There is no explicit branching strategy. When the code has been tested, it is merged to master. The master branch is used for application releases to production.

The database is strictly code first. All changes are done with migration files.

3.3 Team

The development organization is agile. The team composition is outlined in the table below.

Role	FTE	Experience in team
Architect Developer	0.7	Senior, 5 years
Scrum master Project lead Product owner Tester	0.7	Senior, 3 years
Designer Front end developer Tester	0.8	Senior, 5 years
Designer Tester	0.7	Senior, 5 years

Users are requested to do intermediate testing and acceptance testing. On occasion one of the NZa testers can test the application.

3.4 Plan

New features are recorded on the backlog in GitLab. Features are a result of wishes of both developers, users and stakeholders of the application. When new features are requested these are discussed with all parties involved.

Major bugs are picked up at the beginning of sprints. Minor bugs are picked up if they relate to the new functionality that is being developed, or when there is time for the development team to pick up extra work.

3.5 Build

There is a PowerShell script to build the application for release to either the test, acceptation or production environment.

According to the developers, their current GitLab server is not powerful enough to build the application. Therefore the application is built on a development server instead of using GitLab pipelines.

3.6 Test

The application is tested manually by the developers and by end users. According to the development team there is not enough test capacity of dedicated testers in the organisation, most testing is done by the end users. The end users do not have a lot of time for testing, they usually only test the happy path. Most bugs are found by end users when the application is released to production.

There is no test framework and there are no unit tests or integration tests.

3.7 Documentation and knowledge preservation

There is a lot of documentation with respect to the architecture of the application and the patterns that are used. This is a recent development, in the past this was not documented, so a lot of documentation is missing. The development team is aware of this and they try to expand and improve the documentation whenever they have time, but there is no explicit policy for documentation.

The user interface is dependent on highly specific and specialised domain knowledge.

3.8 Third-party dependencies

DevExpress is used for custom WPF components in the user interface of the client application.

Operation

4 Operation

4.1 Introduction

This section provides insight in the operation team composition and process. Furthermore, deployment, operating and monitoring the solution are addressed according to the stages defined in the Operation cycle of the DevOps Lifecycle.

4.2 Process

The database server is managed by NZa's IT partner. Back-ups of the database are made on a daily basis.

When there are bugs in the software, the development team will have to fix those. There is no separate support team.

4.3 Team

There is no operations team. There is one person at NZa that does server maintenance.

Servers are maintained by a third party IT partner.

4.4 Deployment

There is a PowerShell script to release the application to either the test, acceptance, or production environments.

Developers can use the script to release to the test environment. Administrators can use the script to release to acceptance or production.

After release, the database migration scripts are executed automatically.

The database cannot be built from just the migration scripts; views, stored procedures, functions and triggers need to be restored separately.

4.5 Monitoring

There is no monitoring in place.

4.6 Back-up and recovery

There are daily back-ups of the database, but those back-ups are not tested.

Code Quality

yieldD

5 Code Quality

5.1 Introduction

This section provides insights in the quality of the code as researched by the YieldDD experts. Each application discussed has been subjected to our qualitative review process.

The following chapters evaluate the code according to the ISO/IEC 25010 standard for system and software quality and result in qualitative assessment of the characteristics of Maintainability and Security.

Maintainability is defined as the degree of effectiveness and efficiency with which a product or system can be modified to improve it, correct it, or adapt it to changes in environment, and in requirements. This characteristic is composed of the sub-characteristics Modularity, Reusability, Analysability, Modifiability and Testability.

To qualify these characteristics the provided source code is analysed by two qualified software development experts and scored as Poor, Poor/Moderate, Moderate, Moderate/Good and Good.

The weighted scores of the analysed aspects result in a qualification of the maintainability and security characteristics. The qualifications used are

- **Above average**: An application is maintainable without the risk of disrupting the business. Code, design, and architecture is clear, changes can be made in one place and tests make sure the application functions as intended.
- **Average:** There is some risk is maintaining the application but in general it is manageable. Tests could be missing, or the design is not clear but not both.
- **Below Average:** There is a significant risk in disruption when maintaining the application. Extending or modifying the code can be proven difficult. Code and functionality are spread out, application structure is complex, tests are missing.

5.2 Characteristics

Two C# solutions have been provided for research. The following tables contain characteristics overviews of the application.

Metric	#	Qualification
Files	1,162	Medium
Lines of source code	70,053	Medium
Lines of executable code	55,375	Medium
Cyclomatic complexity	4,040	High
Depth of inheritance	4	Medium
Class coupling	68	High

Table 4: Code characteristics client (C#)

Metric	#	Qualification
Files	333	Medium
Lines of source code	26,514	Medium
Lines of executable code	20,636	Medium
Cyclomatic complexity	3,611	High
Depth of inheritance	4	Medium
Class coupling	78	High

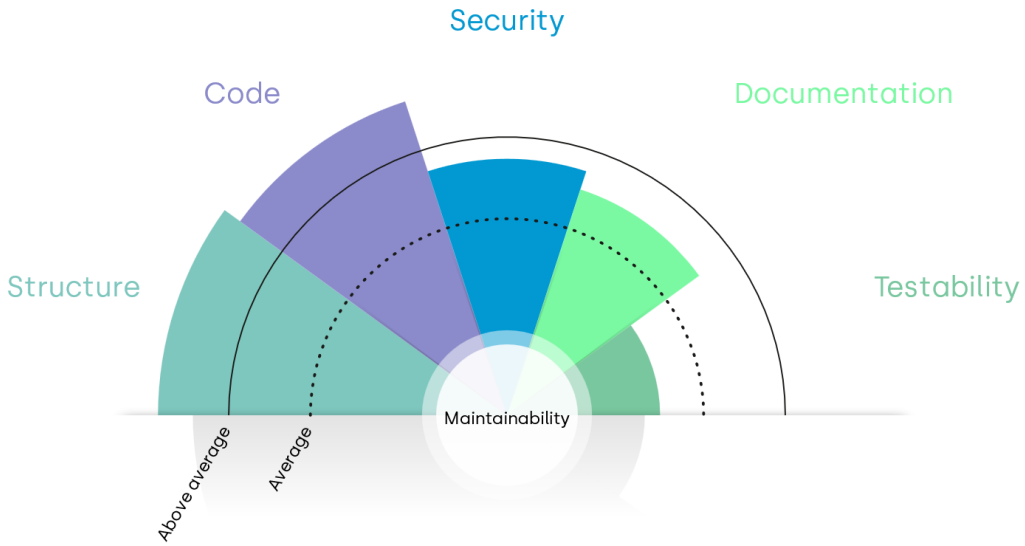
Table 5: Code characteristics server (C#)

Metric	#	Qualification
Files	161	Medium
Lines of source code	36,283	Medium
Lines of executable code	30,233	Medium

Table 6: Code characteristics server (T/Sql)

5.3 Overall

The reporting of the code quality for the application is outlined in the following sections. The code quality is scored on structure, code, security, documentation and testability.



5.4 Structure

The overall solution, project and folder structure is qualified **Above Average**.

The solution is clearly structured into projects and folders and implemented using a Services Oriented Architecture. The client is built using the MVVM pattern and uses DevExpress for custom components.

Source code is managed using Git on a self-hosted Gitlab server.

The framework version is .NET Framework Newer versions of .NET Framework exist. WPF and WCF are also supported for the .NET (Core) stack which is in active development

Class and file names often contain Dutch terms.

Criteria	Score	Impact on
Directory structure	Good	Analysability
Consistent naming	Good	Analysability
Tiers and Layers	Good	Modularity
Dependency management (external - non-framework)	Good	Modularity, Reusability
Dependency management (internal - proprietary libraries)	Good	Modularity, Reusability
Component versioning	N/A	Modularity, Reusability
Source code repositories	Good	Modifiability, Modularity
Language and framework version	Moderate/Good	Modifiability
Code styling	Good	Analysability

5.5 Code

The overall code quality is **Above Average**.

Both client and server applications have a strict folder structure that map to each other. Namespaces are consistent and clear.

The domain is in Dutch and this is reflected by a lot of Dutch terms and variable names in the code.

Most parts of the code are short and simple, but some files are very big. There is deep nesting in places which results in some complexity.

There are magic numbers and strings in the code which make it harder to understand and/or harder to refactor.

Criteria	Score	Impact on
Use of consistent naming conventions	Good	Analysability
Function length	Good	Analysability
Code Length	Moderate/Good	Analysability
Code Complexity	Moderate/Good	Analysability
Single Responsibility Principle	Good	Modifiability
Open Closed Principle	Good	Modifiability, Reusability
Liskov substitution	Good	Modifiability
Interface Segregation	Good	Reusability
Hardcoded strings , numbers	Moderate	Reusability
Procedural vs Declarative	Procedural	Modifiability
Stateful vs Stateless	Stateful	Modifiability, Reusability
Defensive programming	Good	Stability

5.6 Security

The overall security is **Average**.

End users of the application can only access the application in a Citrix session which is authenticated by an Active Directory account. Role-based access control is used, roles are managed in Active Directory. Based on the assigned roles, some parts of the application may or may not be visible to users.

All changes to the decision trees are logged.

There is no automated security tooling like static code analysis or vulnerability testing.

Criteria	Score	Impact on
Security measures	Moderate	Confidentiality, Integrity
Appropriate Authorization/Authentication	Good	Confidentiality, Authenticity
Adequate logging and monitoring	Good	Non-repudiation
Automated security tooling	Poor	Integrity

5.7 Documentation

The overall documentation quality is **Average**.

Architecture decisions are recorded in design documents. There is a fair amount of documentation, but a lot of domain knowledge is not documented and therefore only available as long as the current developers are part of the team.

The code contains comments with references to GitLab ticket numbers as a way to explain why the code is the way it is.

There is no documentation for end users. New users are onboarded by the existing users. There are a few videos showcasing the functionality for new users, but these have not been updated to reflect new functionality or the new interface.

All documentation is in Dutch.

Code comments are in Dutch.

Criteria	Score	Impact on
Descriptive comments	Moderate	Analysability
Inline comments	Moderate	Analysability
Functional documentation	Poor/Moderate	Analysability
Architectural documentation	Good	Analysability

5.8 Testability

The overall testability is **Below Average**.

There are no unit tests and no integration tests.

Writing tests for the business logic is a lot of work. The business logic is grouped together, so writing tests for just the business logic should be possible. Since no tests have been written it is possible that parts of the code are not testable without adaption.

Criteria	Score	Impact on
Code coverage	Poor	Testability
Testability	Moderate	Testability

5.9 Database and T/Sql

The database is strictly code first. Changes are done by expanding the existing SQL files or creating new ones. This is a manual process. Changes are tested in the test or acceptance environments before they are run on the production environment, even in the case of a high priority fix on production.

When a new version of the application is released, the database is updated using the migration files. While the database structure can be rebuilt from the migration files, there is no way to build the full database from scratch. Views, stored procedures, triggers and functions need to be restored separately.

The migration scripts also contain more than 70 stored procedures. Due to the size of the files, it might be hard to find and manage parts of the database.

Software health

yieldD

6 Software Health

6.1 Introduction

The current state of the software is evaluated by scanning and analysing the source code of the application using CAST Highlight. This quantitative analysis provides insight in the software health and benchmarks the solution to put the absolute metrics in perspective.

In CAST Highlight Software Health is defined as an indication on how an application complies with programming best practices that increase resiliency, improve agility and reduce complexity. Software Health score is the straight average of Software Resiliency, Software Agility and Software Elegance scores.

The solution is benchmarked against a large number of applications scanned by CAST Highlight and divided in quartiles based on the scores. These quartiles are qualified as listed in the following table.

Quartile	Qualification
First	Excellent
Second	Good
Third	Moderate
Fourth	Poor

Table 7: Benchmark quartiles and qualifications

Note that the scores can be benchmarked on the dimensions of software health and technology. When the solution is implemented using multiple technologies, the overall score is benchmarked.

To avoid false positives on generated code, third-party dependencies, deployment scripts and such, only produced source code is included in the analysis.

6.2 Front end

Figure 3 below shows the software health for the front end.

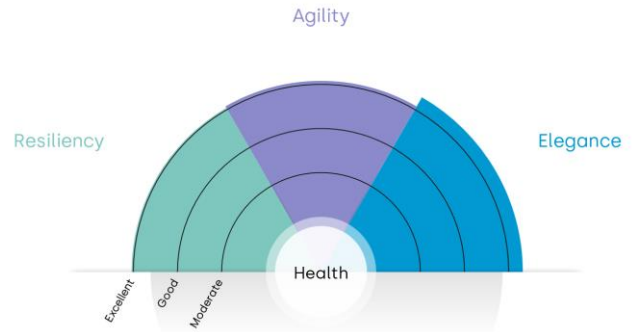


Figure 3: Front end software health

6.3 Back end

Figure 4 and Figure 5 show the software health for the back end.

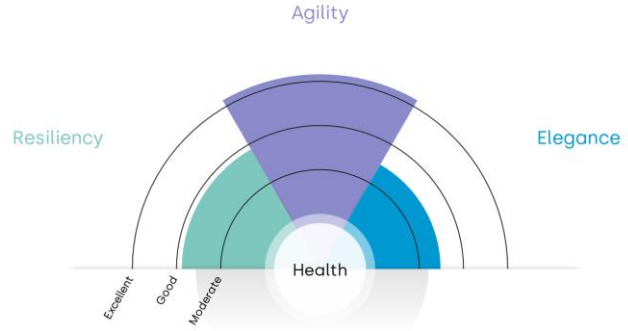


Figure 4: Back end software health (T/Sql)

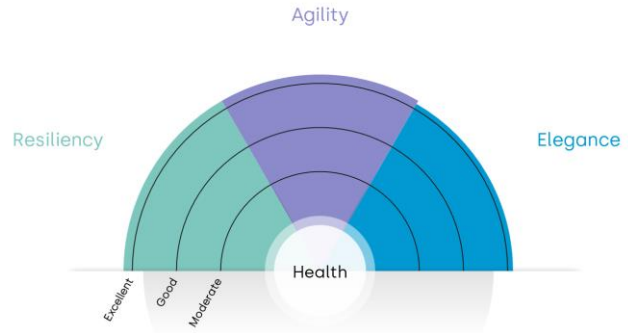


Figure 5: Back end software health (C#)

6.4 Software Resiliency

Indicates programming best practices that make software bullet-proof, more robust and secure. This index is derived from technology-specific code analysis which searches for the presence of code patterns and bad programming practices which may compromise the reliability of the software in the short term. Higher the Software Resiliency, lower is the likelihood of defects to occur in production.

The table below lists the software resiliency scores per dimension.

Dimension	Score	Benchmark
Front End	75.6	Excellent
Back End – C#	80.2	Excellent
Back End – T/Sql	47.2	Moderate

Table 8: Software resiliency per dimension

Most noticeable improvement opportunities are:

C#

- Avoid literal numbers (i.e. magic numbers are not so magic).
- Using { curly braces } is less error-prone.

T/Sql

- RETURN should be used to terminate a database query or a stored procedure.
- The code contains too many functions and procedures doing an Insert, Update or Delete without managing a transaction.

6.5 Software Agility

Indicates the easiness of a development team to understand and maintain an application. This index is derived from technology-specific code analysis which searches for the presence of embedded documentation and code readability good practices.

The table below lists the software agility scores per dimension.

Dimension	Score	Benchmark
Front End	76.6	Excellent
Back End – C#	79.7	Excellent
Back End – T/Sql	79.1	Excellent

Table 9: Software agility per dimension

Most noticeable improvement opportunities are:

C#

- Ensure the code contains enough blocks of comments.
- Operators and operands should use appropriate spacing to help readability.
- Short code identifiers are harder to understand.

T/Sql

- The code contains too many functions and procedures parameters that are not commented.
- A line of code should not be too long to help readability.

6.6 Software Elegance

Measures the ability to deliver software value with less code complexity. A low Software Elegance score indicates decreased quality in the code resulting in higher defects which may become costly to fix in the mid-term. Software elegance also measures how easy or hard it is to read and understand code.

The table below lists the software elegance scores per dimension.

Dimension	Score	Benchmark
Front End	82.6	Excellent
Back End – C#	77.7	Excellent
Back End – T/Sql	37.1	Moderate

Table 10: Software elegance per dimension

Most noticeable improvement opportunities are:

C#

- Structural code complexity may be too high.
- Bulky files are complex to work with.

T/Sql

- The code contains too many DDL and DML interleaving.
- Bulky files are complex to work with.


yieldDD
software due diligence;
yieldDD.com
info@yieldDD.com
085 08 663 28