CODESCENE

Enterprise Edition

1.5.2

December 20, 2016

Contents

Welcome to the CodeScene documentation!

This documentation is divided into sections, each being suited for different types of information you might be looking for.

- *Getting Started* (page 3) helps you take the first steps after you purchase of CodeScene. You will learn how to install and setup the tool, as well as running your first analysis.
- *Guides* (page 12) walk you through specific features and aspects of the tool, focusing on how you can use them to achieve certain goals.
- *Configuration* (page 58) explains how you configure projects to get the best possible analysis results.

Chapter 1

Getting Started

CodeScene is a web-based application that you install on a server and access via your web browser. Once you've installed the tool, you will be up and running with your first analysis results in just a few minutes.

1.1 Configure Your Environment

CodeScene runs anywhere a modern Java Virtual Machine (JVM) runs. We test the tool on Mac OS, Windows, and different Linux distributions.

The system requirements depend upon the size and history of the codebase you want to analyze. In general, RAM memory is the most critical resource on the server. That means you want to ensure that there's at least 4 Gb of RAM available for the CodeScene application.

1.1.1 Install the Supporting Tools

You need to install the following to run CodeScene:

- A Java run-time (or JDK if you run from the command prompt), 64-bit version, *at least Java 1.8*. You ensure you have the right Java version by typing java -version in a command prompt.
- Have a Git client on your path since the tool will assume there's an executable named "git" somewhere. Your Git client has to be *at least version 2.6*. You ensure you have the right Git client version by typing git –version in a command prompt.

Please note that you can specify a custom Git client in the Configuration section once you login to CodeScene.

1.1.2 Setup an SSH Key for Git

CodeScene operates on local clones of your Git repositories. CodeScene does an automated *git pull* before an analyses, which lets you see the latest changes reflected in your analysis results. This means you need to grant CodeScene access to your repository origins. You do that by providing an SSH key (see for example https://git-scm.com/book/be/v2/Git-on-the-Server-Generating-Your-SSH-Public-Key).

NOTE: If you chose to run CodeScene in Tomcat, the SSH key has to be associated with the Tomcat user since that's the user who will access the Git repositories.

1.2 Installation

1.2.1 Run CodeScene from the Command Line

We typically deploy CodeScene as a service in an application container (e.g. Tomcat) as described in the next session. However, the simplest way to get the application up and running is by launching the standalone JAR:

```
java -jar codescene.standalone.jar
```

This will launch a web application that listens on port 3003 (you can override that by setting a different port through the environment variable `CACS_RING_PORT`.

Now, please point your web browser to *localhost:3003*.

1.2.2 Install CodeScene on a Server

A server installation is the recommended way of running CodeScene. This section described how you do that using Tomcat.

Run CodeScene in Tomcat

CodeScene is delivered as a WAR file (**W**eb application **AR**chive). We recommend that you deploy it using Tomcat (https://tomcat.apache.org/index.html).

Specify a file folder for the database

CodeScene uses an embedded database. That means, you don't have to install any database or drivers yourself. However, you need to specify a path to a file folder where CodeScene is allowed to store its database. Here's how you configure Tomcat to do that:

1. Open the file `context.xml` located under the conf directory in your Tomcat installation.
2. Add an `<Environment>` tag to context.xml that specifies the path to a folder you want to use for the database (see the example below).
3. Save context.xml.

Here's an example on how context.xml may look on a Windows installation (note that you need to modify the path to fit your environment):

```
<Context>
  <WatchedResource>WEB-INF/web.xml</WatchedResource>
  <WatchedResource>${catalina.base}/conf/web.xml</WatchedResource>
  <Environment name="empear.dbpath"
               value="C:\\some\\path\\to\\the\\database\\empear.codescene"
               type="java.lang.String"/>
</Context>
```

In case you run on a Linux-based system, you just specify a different path format. For example:

```
<Context>
  <WatchedResource>WEB-INF/web.xml</WatchedResource>
  <WatchedResource>${catalina.base}/conf/web.xml</WatchedResource>
  <Environment name="empear.dbpath"
               value="/Users/adam/Documents/Empear/deployment/empear.codescene"
               type="java.lang.String"/>
</Context>
```

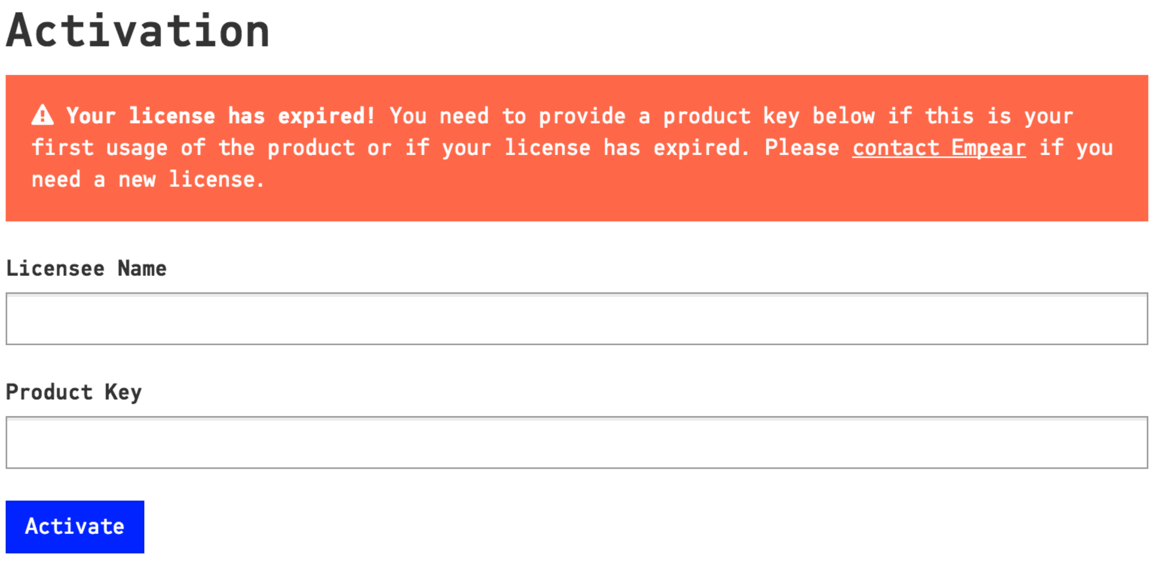
NOTE: Please ensure that Tomcat has write access to the folder you specify.

Deploy the codescene.war

Once Tomcat is up and running, with your modified `context.xml`, you just copy the `codescene.war` to the webapps folder in your Tomcat installation.

Access CodeScene

By default, Tomcat will launch CodeScene on port 8080 and at the path */codescene/*. If you're logged in on the server, you access the application on http://localhost:8080/codescene/login. You should see the activation screen in your web browser (see Fig. 1.1).


Activation

Your license has expired! You need to provide a product key below if this is your first usage of the product or if your license has expired. Please contact Empear if you need a new license.

Licensee Name

Product Key

Fig. 1.1: The first time your login you are prompted to activate the application.

Enter the credentials you received in your license file. You're now ready to login (see Fig. 1.2).

The first time you login, you use *the same* credentials to login as you used to activate the application. That is, give your Licensee Name as User Name and your Product Key as Password.

You're now up and running with CodeScene!

1.2.3 Configure additional users

You are granted administration privileges each time you login with your license credentials (note that you can do that at any time, for example to administrate users).

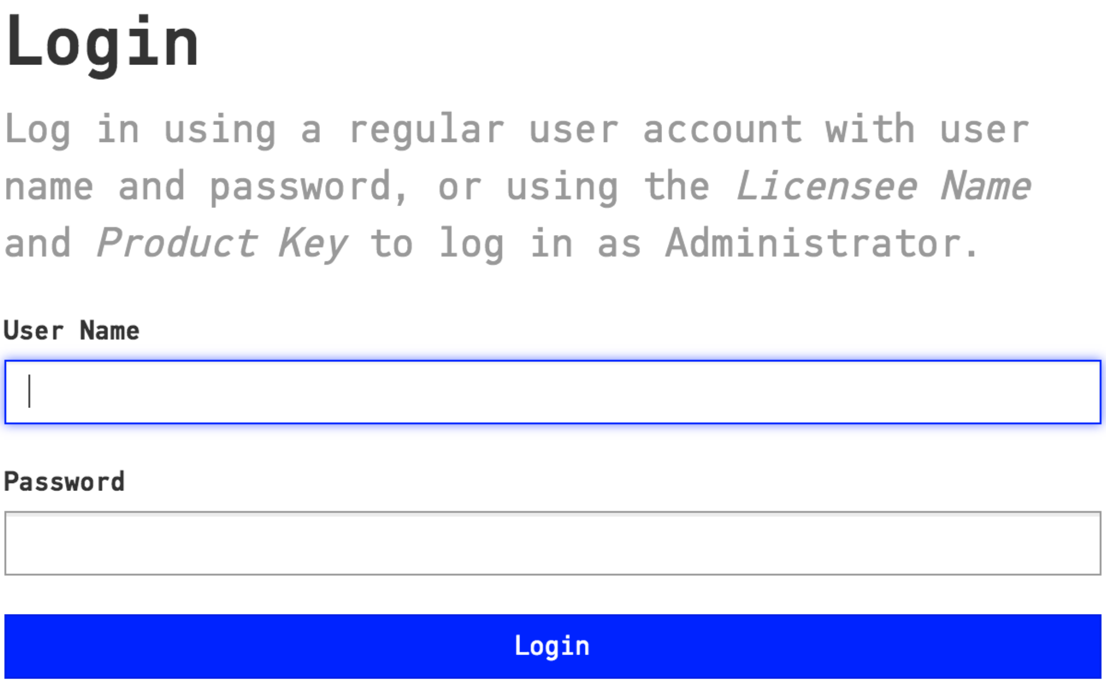
You can add new users and assign them roles in the global configuration. *Users and Roles* (page 69) describes this in greater detail.

1.3 Run an Analysis

1.3.1 Creating a New Project

Your first step is to create and configure a project. You do that by clicking on the "Create New Project" button (see Fig. 1.3).

Once you click the "Create New Project" button you are prompted with a choice (see Fig. 1.4). Select *Specify Paths* if you plan to analyze just one or two repositories and enter the paths manually. Select *Scan Directory* to auto-import multiple repositories into your analysis project.



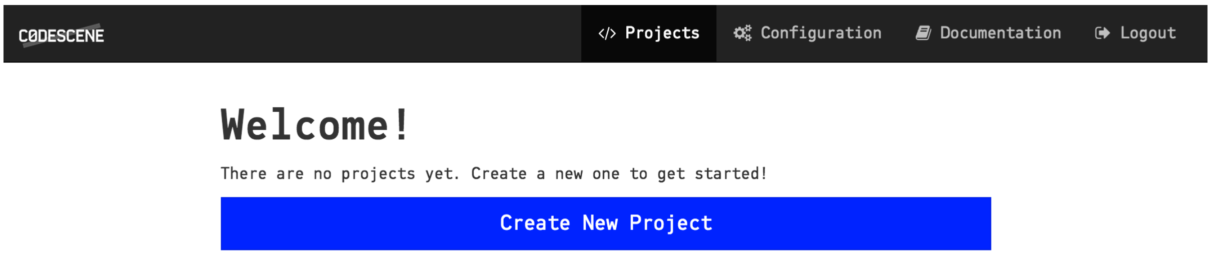
Login

Log in using a regular user account with user name and password, or using the *Licensee Name* and *Product Key* to log in as Administrator.

User Name

Password

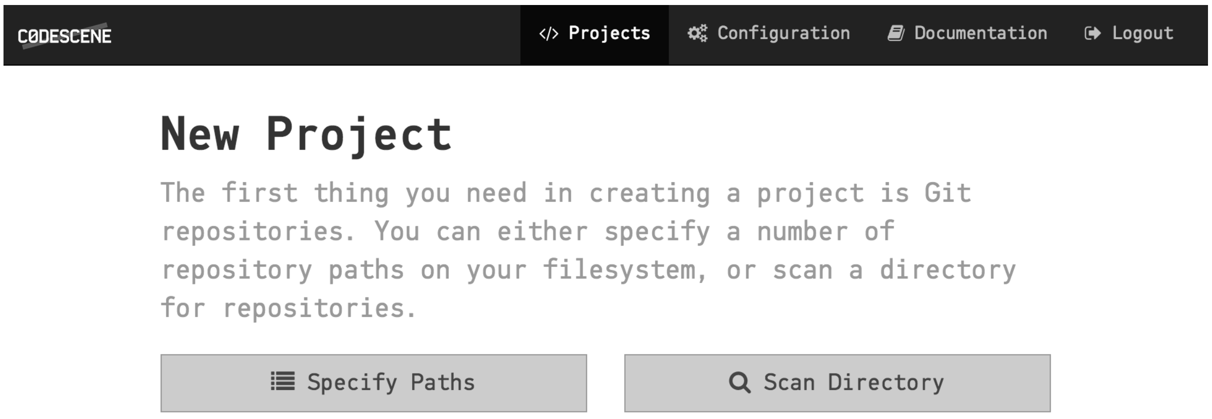
Fig. 1.2: Once you've activated the tool you're ready to login.



Welcome!

There are no projects yet. Create a new one to get started!

Fig. 1.3: Click on the "Create New Project" button to create a project and configure it for analysis.



New Project

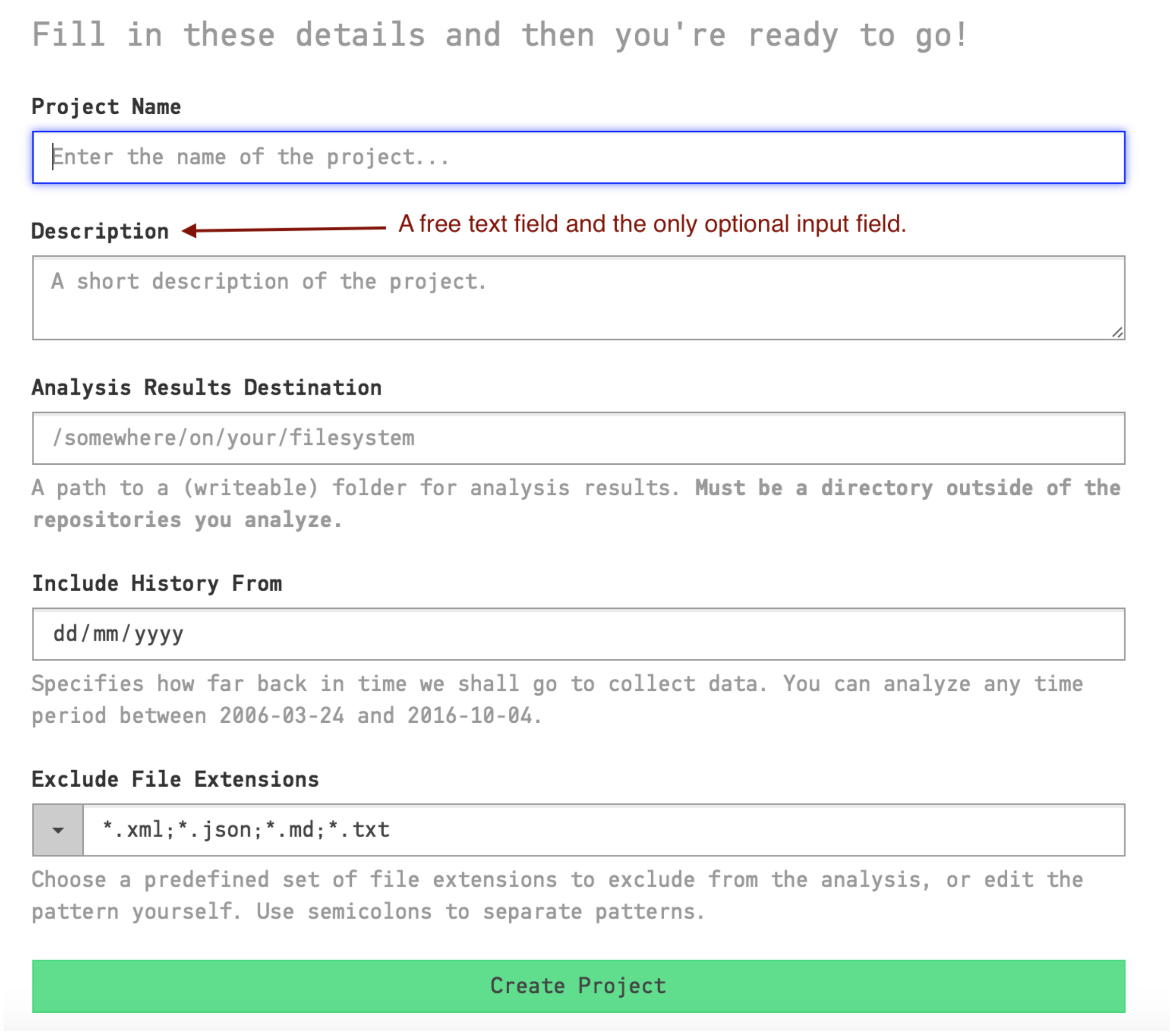
The first thing you need in creating a project is Git repositories. You can either specify a number of repository paths on your filesystem, or scan a directory for repositories.

Fig. 1.4: Specify the local Git repository paths.

If you chose to *Specify Paths*, just type (or copy-paste) the path to your local Git repository clones. You can add as many repositories as you need.

Once you click "Continue", you arrive at the "Project Details" page (see Fig. 1.5). There are a number of important configuration options in this step. The *Configuration* (page 58) include advice on how you select an analysis period. When in doubt, specify the earliest possible starting date as indicated in the help text.

NOTE: It's important that the Analysis Results file folder that you specify is writable for the Tomcat user; all analysis result content will be stored there.



Fill in these details and then you're ready to go!

Project Name

Enter the name of the project...

Description

A free text field and the only optional input field.

A short description of the project.

Analysis Results Destination

/somewhere/on/your/filesystem

A path to a (writeable) folder for analysis results. **Must be a directory outside of the repositories you analyze.**

Include History From

dd/mm/yyyy

Specifies how far back in time we shall go to collect data. You can analyze any time period between 2006-03-24 and 2016-10-04.

Exclude File Extensions

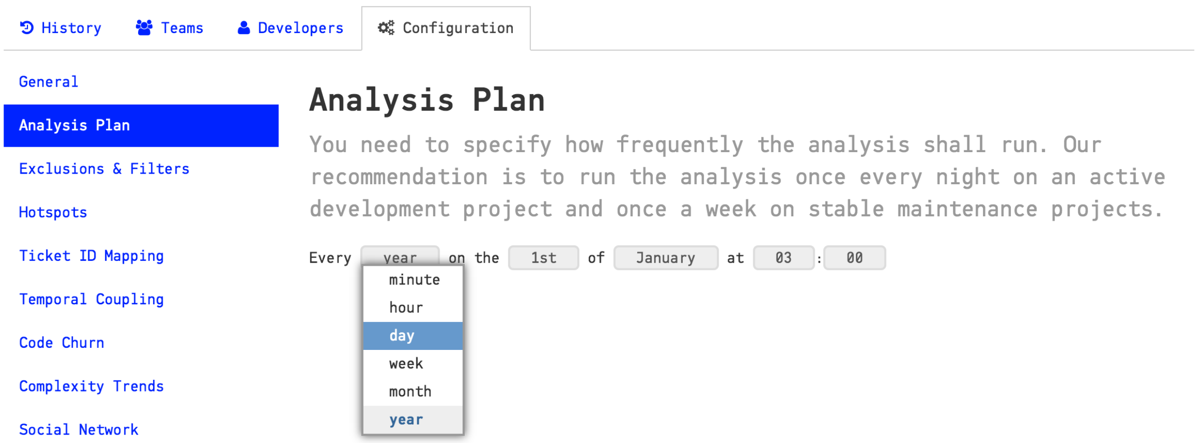
.xml;.json;*.md;*.txt

Choose a predefined set of file extensions to exclude from the analysis, or edit the pattern yourself. Use semicolons to separate patterns.

Fig. 1.5: The detailed configuration lets you specify analysis period and a result path.

Once you've created the project you'll arrive at its configuration details. And yes, there's a lot, really a lot, of configuration parameters. The good news are that you normally don't have to change any of these parameters since they all have sensible defaults. However, you want to look at your Analysis Plan. Go to the "Analysis Plan" configuration as shown in Fig. 1.6 and specify a suitable interval, for example once every night.

From now on, CodeScene will run all analyses automatically according to your plan. However, you probably don't want to wait for the next scheduled run to get results on your codebase. That's why CodeScene supports a forced analysis as described in the next section.



Analysis Plan

You need to specify how frequently the analysis shall run. Our recommendation is to run the analysis once every night on an active development project and once a week on stable maintenance projects.

Every year on the 1st of January at 03 : 00

Fig. 1.6: Your analysis plan specifies how often an analysis is run.

1.3.2 Force an Analysis

CodeScene lets you run an analysis on demand. Just go to the dashboard and press the *Run* button as illustrated in Fig. 1.7.

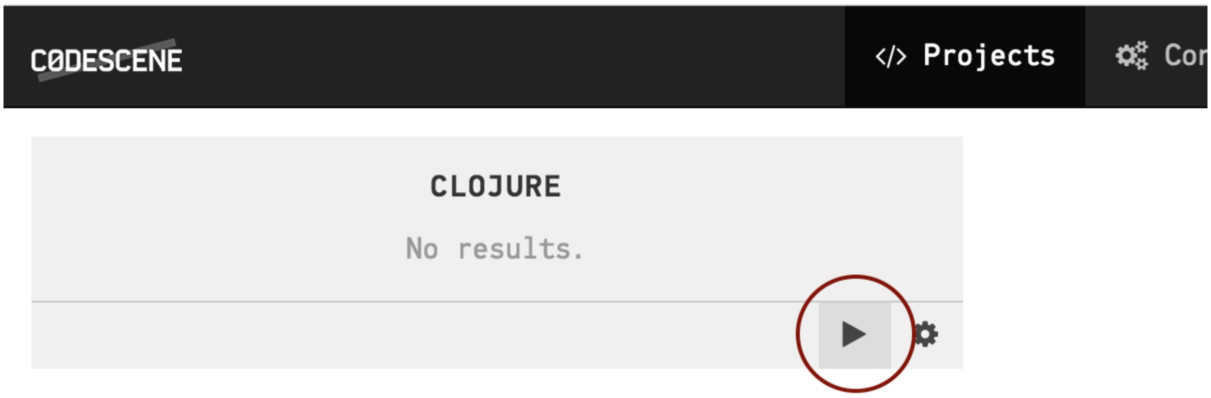


Fig. 1.7: Press the Run button to force an analysis.

1.3.3 Run a Retrospective

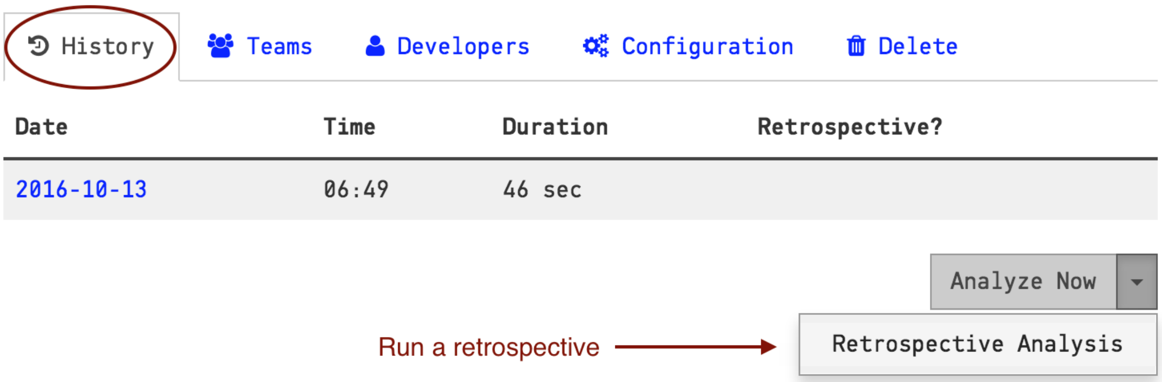
CodeScene also includes the option to run an analysis tailored to a *Retrospective*. This feature is located on the “History” tab of your analysis project as illustrated in Fig. 1.8.

For a detailed description of the use cases for Retrospectives, read the article The Happy Marriage of Retrospectives and Software Evolution.

1.3.4 Find your Way Around

We’ve worked to make CodeScene as easy as possible for you to use. Basically, you just need to remember three things:

1. Click the *cogs button* of your project (see Fig. 1.9) to access details, configuration, and to force analyses.
2. Click on the Latest Analysis link to inspect your analysis results (see Fig. **??**).
3. Click on the “CODESCENE” text to return to the main screen, should you ever get lost.



Date	Time	Duration	Retrospective?
2016-10-13	06:49	46 sec	

Fig. 1.8: *A retrospective lets you analyze the development activity in the past sprint/iteration.*

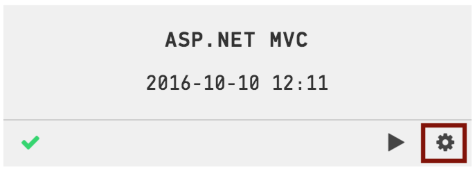
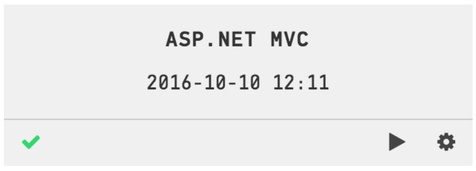
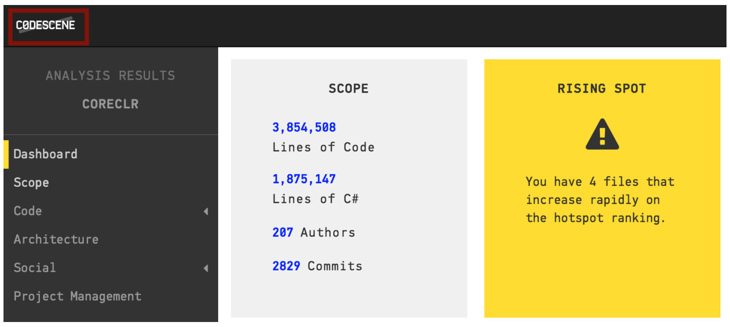


Fig. 1.9: *The cogs button in the project tile takes you to the project details and configuration.*





1.4 Ensure you use a Mailmap

The social metrics need to identify each developer that contributes code. Unfortunately, it's common that developers have multiple Git aliases, which will bias the social metrics.

You avoid that bias by providing a `.mailmap`. A `.mailmap` is a Git feature. It's a file that specifies a mapping from multiple aliases to one for each developer. CodeScene automatically uses the `.mailmap` when it's present.

Read the Git Documentation on mapping authors for a description on how to configure the `.mailmap`.

1.5 Use a Reverse Proxy for HTTPS Support

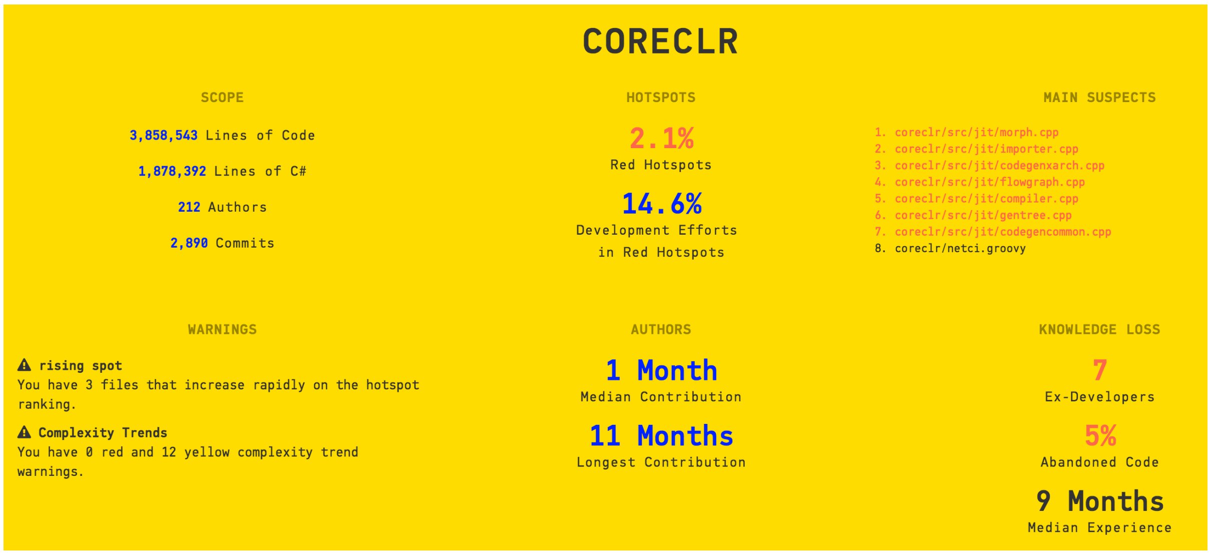
CodeScene doesn't implement HTTPS support itself. Instead we recommend that you put a reverse proxy in front of the application if you need SSH. We recommend Nginx as reverse proxy.

1.6 Display A Monitor Dashboard

Use CodeScene's monitor view to display an auto-updated dashboard with the status of your codebase.

1.6.1 View the Monitor Dashboard

CodeScene presents a high-level monitor view that displays the key metrics in your codebase (see Fig. 1.10). Present it on a TV or a big screen in the office and share the automatic updates with your team.



CORECLR

SCOPE

3,858,543 Lines of Code

1,878,392 Lines of C#

212 Authors

2,890 Commits

HOTSPOTS

2.1%
Red Hotspots

14.6%
Development Efforts
in Red Hotspots

MAIN SUSPECTS

1. coreclr/src/jit/morph.cpp
2. coreclr/src/jit/importer.cpp
3. coreclr/src/jit/codegenxarch.cpp
4. coreclr/src/jit/flowgraph.cpp
5. coreclr/src/jit/compiler.cpp
6. coreclr/src/jit/gentree.cpp
7. coreclr/src/jit/codegencommon.cpp
8. coreclr/netci.groovy

WARNINGS

⚠ **rising spot**
You have 3 files that increase rapidly on the hotspot ranking.

⚠ **Complexity Trends**
You have 0 red and 12 yellow complexity trend warnings.

AUTHORS

1 Month
Median Contribution

11 Months
Longest Contribution

KNOWLEDGE LOSS

7
Ex-Developers

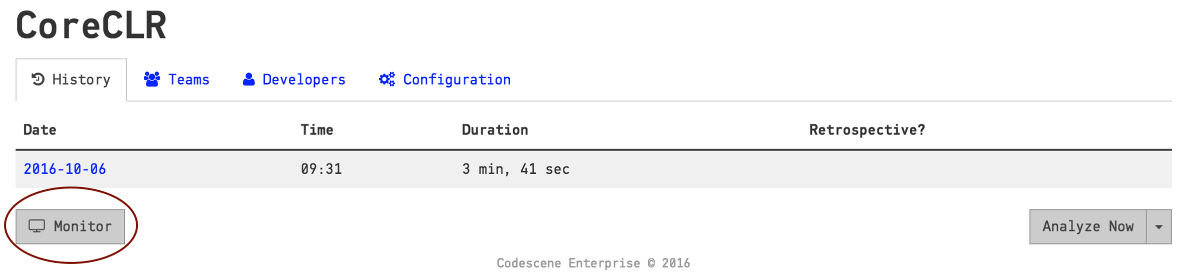
5%
Abandoned Code

9 Months
Median Experience

Fig. 1.10: The monitor dashboard gives you a high-level overview of your codebase.

The monitor dashboard is automatically updated with the latest analysis results.

You access the monitor dashboard from the "History" view of your project configuration (see Fig. 1.11). Please note that you need to have the role "Full Read-only Access" to view the dashboard, so please create a dedicated user for the monitoring as described in *Users and Roles* (page 69).



CoreCLR

Date	Time	Duration	Retrospective?
2016-10-06	09:31	3 min, 41 sec	

Codescene Enterprise © 2016

Fig. 1.11: Access the monitor dashboard from the History view in the project configuration.

Chapter 2

Guides

These guides walk you through specific features and aspects of CodeScene Enterprise Edition. They are divided into *Technical*, *Architectural*, and *Social* guides.

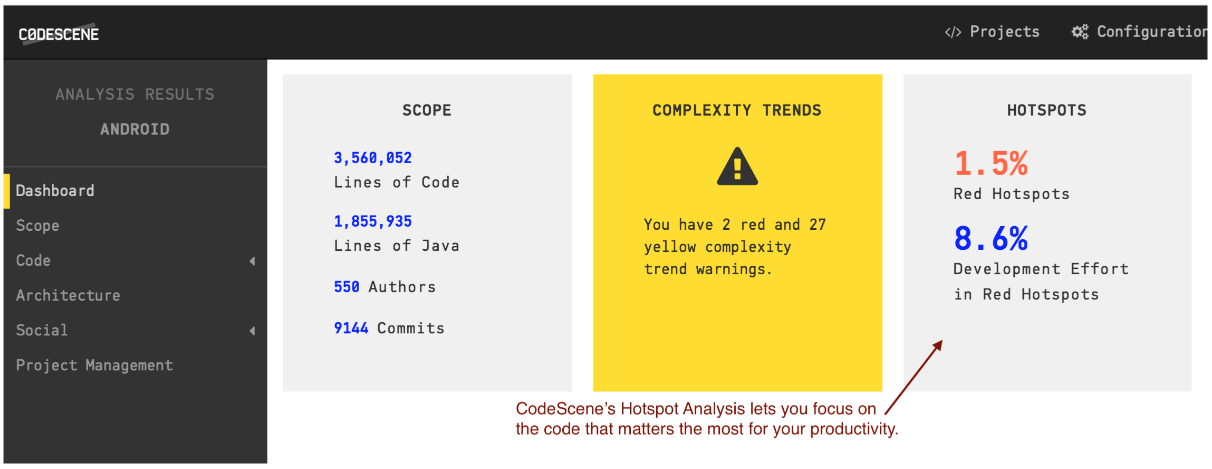
2.1 Technical

2.1.1 Hotspots

Hotspots are the workhorse of software analyses and our recommended starting point as you explore your codebase.

What is a Hotspot?

Your development activity tends to be located to relatively few modules as illustrated in Fig. 2.1. A Hotspot analysis helps you identify those modules where you spend most of your time. This is information you use to improve the parts that really matter. The parts where you're likely to get a return on your investment.



3,560,052 Lines of Code

1,855,935 Lines of Java

550 Authors

9144 Commits

You have 2 red and 27 yellow complexity trend warnings.

1.5% Red Hotspots

8.6% Development Effort in Red Hotspots

CodeScene's Hotspot Analysis lets you focus on the code that matters the most for your productivity.

Fig. 2.1: The dashboard gives you a high-level overview of the Hotspot activity in your code.

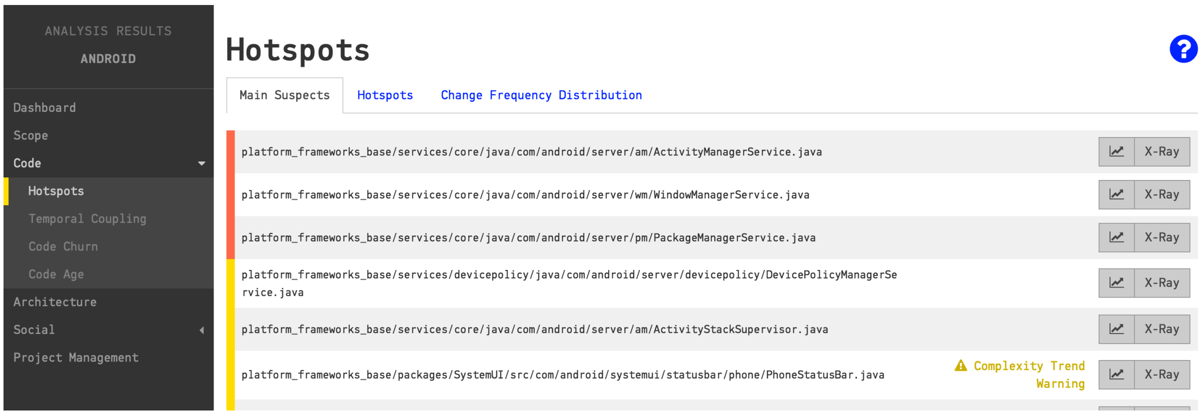
A hotspots is *complicated code that you have to work with often.*

Focus on your Main Suspects

To prioritize your hotspots, CodeScene employs algorithms that look at deeper change patterns in the analysis data. The rationale is that complicated code that changes often is more of a problem if:

1. The hotspot has to be changed together with several other modules.
2. The hotspot affects many different developers on different teams.
3. The hotspot is likely to be a coordination bottleneck for multiple developers.

This algorithm allows CodeScene to rank and prioritize the hotspots in your codebase as illustrated in Fig. 2.2.



```
platform_frameworks_base/services/core/java/com/android/server/am/ActivityManagerService.java
platform_frameworks_base/services/core/java/com/android/server/wm/WindowManagerService.java
platform_frameworks_base/services/core/java/com/android/server/pm/PackageManagerService.java
platform_frameworks_base/services/devicepolicy/java/com/android/server/devicepolicy/DevicePolicyManagerSe
rvice.java
platform_frameworks_base/services/core/java/com/android/server/am/ActivityStackSupervisor.java
platform_frameworks_base/packages/SystemUI/src/com/android/systemui/statusbar/phone/PhoneStatusBar.java
```

Fig. 2.2: CodeScene prioritizes the Hotspots in your code.

The red category 1 hotspots are the ones you want to focus on improving first. Improvements to those parts are likely to give you a large return on your investment.

Once you've addressed those hotspots, the yellow category 2 hotspots become interesting as well. A category 2 hotspot is likely to be a real problem as well, albeit not as severe as category 1.

Shrink the Problem Space with Main Suspects

The ranked list presented as the *Main Suspects* is based on probabilities; We cannot guarantee that the code represents a true problem. But it's likely to be one. And, best of all, that data is based on how your developers have worked with the system so far.

The main advantage of using the *Main Suspects* as a guide to improvements is that you're able to narrow down refactorings to a small part of the system. That in turn will give you more time to tackle larger issues once you've made these initial improvements.

Explore the Hotspot Activity

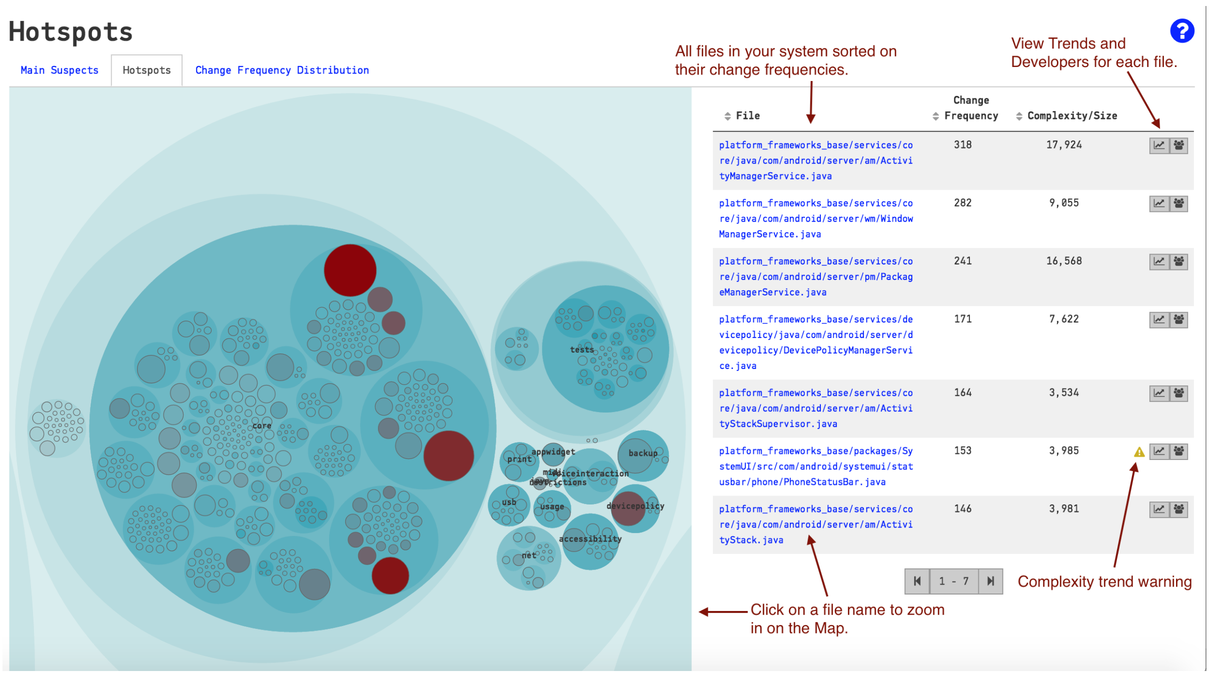
CodeScene also lets you explore the overall Hotspot activity in your code. These Hotspots are calculated from two different data sources:

1. We use the lines of code in each file as a proxy for complexity.
2. We use the change frequency of each file as a proxy for the effort you've spent on that code.

You want to look for an overlap between the two metrics. That's why CodeScene presents you with a sorted table of your hotspots. Fig. 2.3 shows an example from the Android codebase.

From here you can:

- Click on a hotspot in the table to zoom in on it in the Hotspots Activity Map.
- Click outside the hotspot to zoom out.


Hotspots
Main Suspects
Hotspots
Change Frequency Distribution
All files in your system sorted on their change frequencies.
View Trends and Developers for each file.
Complexity trend warning
Click on a file name to zoom in on the Map.
1 - 7
core
tests
appwidget
print
backup
deviceinteraction
restrictions
usb
usage
devicepolicy
accessibility
net

File	Change Frequency	Complexity/Size	
platform_frameworks_base/services/core/java/com/android/server/am/ActivityManagerService.java	318	17,924	
platform_frameworks_base/services/core/java/com/android/server/wm/WindowManagerService.java	282	9,055	
platform_frameworks_base/services/core/java/com/android/server/pm/PackageManagerService.java	241	16,568	
platform_frameworks_base/services/devicepolicy/java/com/android/server/devicepolicy/DevicePolicyManagerService.java	171	7,622	
platform_frameworks_base/services/core/java/com/android/server/am/ActivityStackSupervisor.java	164	3,534	
platform_frameworks_base/packages/SystemUI/src/com/android/systemui/statusbar/phone/PhoneStatusBar.java	153	3,985	⚠
platform_frameworks_base/services/core/java/com/android/server/am/ActivityStack.java	146	3,981	

Fig. 2.3: Hotspots in the Android codebase.

- View the complexity trend of the Hotspot by clicking on the trend buttons (see :doc: *guides/technical/complexity-trends*).

Explore your Hotspots

A large codebase may contain many different hotspots. You will also notice clusters of hotspots, which may indicate that a whole component or package is undergoing heavy changes.

The Hotspots Activity Map in CodeScene lets you explore your whole codebase interactively as illustrated in Fig. 2.4.

The hotspots map is interactive and hierarchical; Each large blue circle represents a folder in your codebase. That means you can zoom in and out to the level of detail you're interested in:

- Click on one of the large, blue circles representing a directory to zoom in on its content.
- Click on a Hotspot to view information about it and to access its context menu to run detailed analyses.
- Click outside the circle representing a zoomed in folder to zoom out again.
- Hover the mouse over a circle to see information about the module it represents.

The most common interaction is to click on a Hotspot to get more details about it as illustrated in Fig. 2.5

Let's look at some use cases now that you know how the Hotspots analysis works.

Know how to use Hotspots

A Hotspot Map has several use cases and also serves multiple audiences like developers and testers:

- *Developers use hotspots to identify maintenance problems.* Complicated code that we have to work with often is no fun. The hotspots give you information on where those parts are. Use that information to prioritize re-designs.

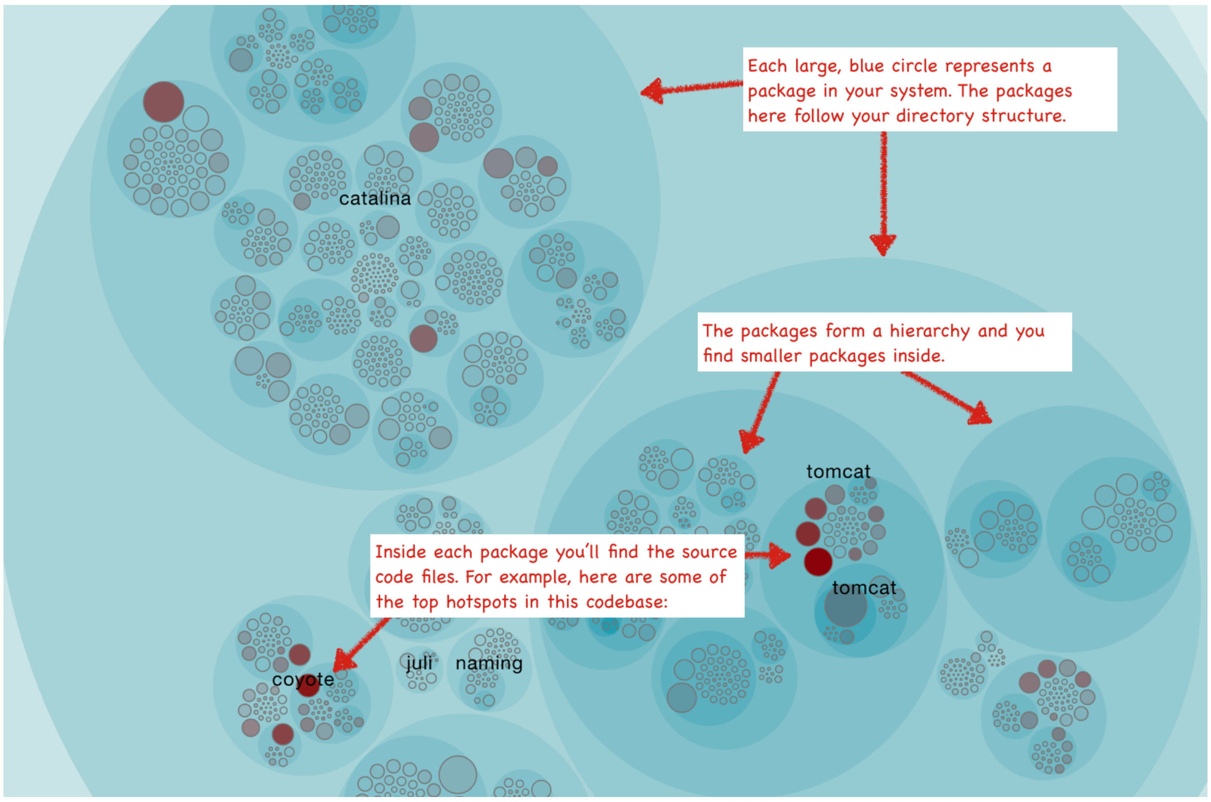
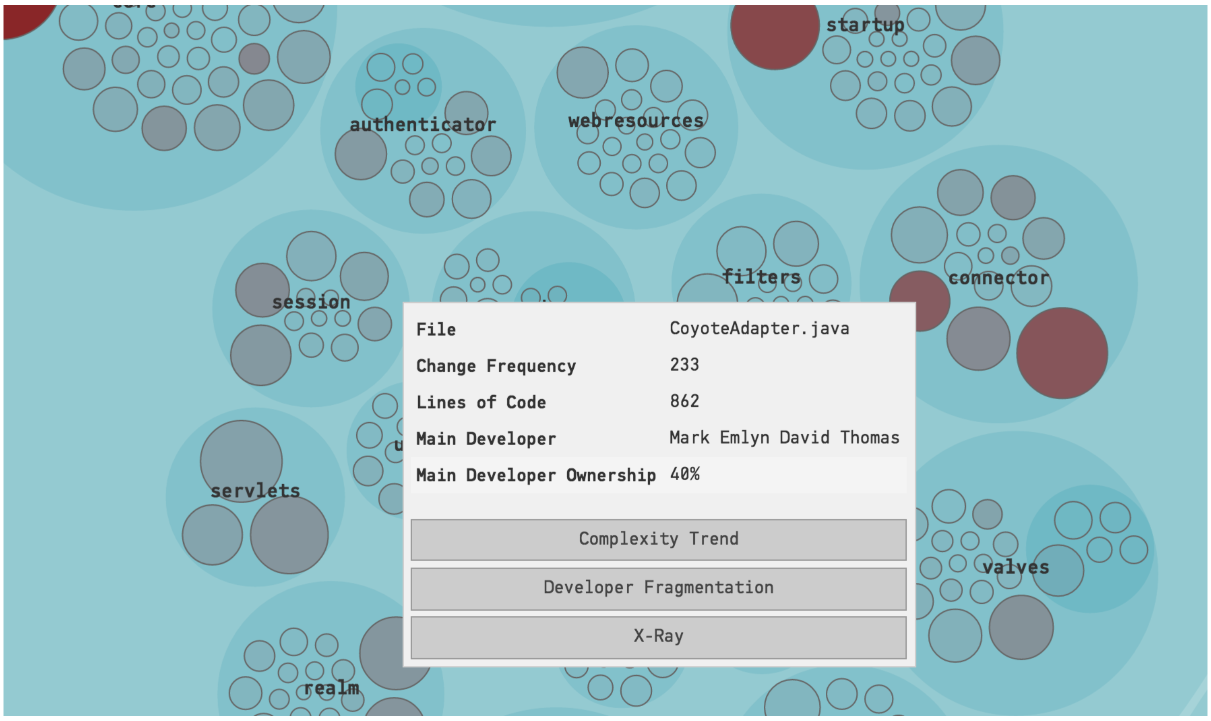

Each large, blue circle represents a package in your system. The packages here follow your directory structure.
The packages form a hierarchy and you find smaller packages inside.
Inside each package you'll find the source code files. For example, here are some of the top hotspots in this codebase:
catalina
tomcat
tomcat
coyote
juli
naming

Fig. 2.4: Hotspots show you the activity in your codebase.



```
File                      CoyoteAdapter.java
Change Frequency          233
Lines of Code             862
Main Developer            Mark Emlyn David Thomas
Main Developer Ownership  40%
```

Fig. 2.5: Click on a Hotspot to access the context menu.

- *Hotspots points to code review candidates.* At Empear we're big fans of code reviews. Code reviews are also an expensive and manual process so we want to make sure it's time well invested. In this case, use the hotspots map to identify your code review candidates.
- *Hotspots are input to exploratory tests.* A Hotspot Map is an excellent way for a skilled tester to identify parts of the codebase that seem unstable with lots of development activity. Use that information to select your starting points and focus areas for exploratory tests.

Use Hotspots in your Daily Work

How well does Hotspots work in practice? Well, it turns out there's strong scientific support behind the metric. The research has often focused on bug predictions, which is relevant since bugs are one of the main issues behind expensive software maintenance.

The book "Your Code as a Crime Scene" (Tornhill, 2015) dives deeper into those research findings to explain why and how Hotspots work. But let's just summarize the conclusions in one line: There's a strong correlation between Hotspots, maintenance costs and software defects. Hotspots are an excellent starting point if you want to find your productivity bottlenecks in code.

That means you want to take your Hotspots seriously. Our recommendation is to run a Hotspot analysis at least once a week. It's also a good idea to share your findings with your team. Why not gather everyone around a Hotspot Map every now and then?

2.1.2 Temporal Coupling

Temporal Coupling means that two (or more) modules change together over time. Exploring Temporal Coupling in our codebases often gives us deep and unexpected insights into how well our designs stand the test of time.

Understand Temporal Coupling

CodeScene provides several different metrics for temporal coupling. The tool considers two modules coupled in time:

- if they are modified in the same commit, or
- if they are modified by the same programmer within a specific period of time, or
- if they refer to the same Ticket ID in their commit messages.

The temporal coupling graph in CodeScene shows a hierarchical view of your temporal coupling. Hover over a label in the graph to highlight its dependants as illustrated in Fig. 2.6.

The initial graph is great to spot interesting temporal dependencies (we'll discuss them soon). CodeScene also presents a tabular view of the temporal coupling in your system, as illustrated in Fig. 2.7.

Coupled Entities Two files that tend to change together over time.

Degree of Coupling How often the files change together. The first pair in Fig. 2.7 change together 74% of the time.

Average Revisions This measure is used to filter out temporal couples that don't pass a configurable threshold. We do not want to consider two files coupled just because they were created in the same commit.

In this guide you'll see just how powerful Temporal Coupling is. The more experience we get with the analysis, the more use cases there seem to be. For example, you'll learn to use the Temporal Coupling results to:

- Detect software clones (aka copy-paste code).
- Evaluate the relevance your unit tests.
- Detect architectural decay.

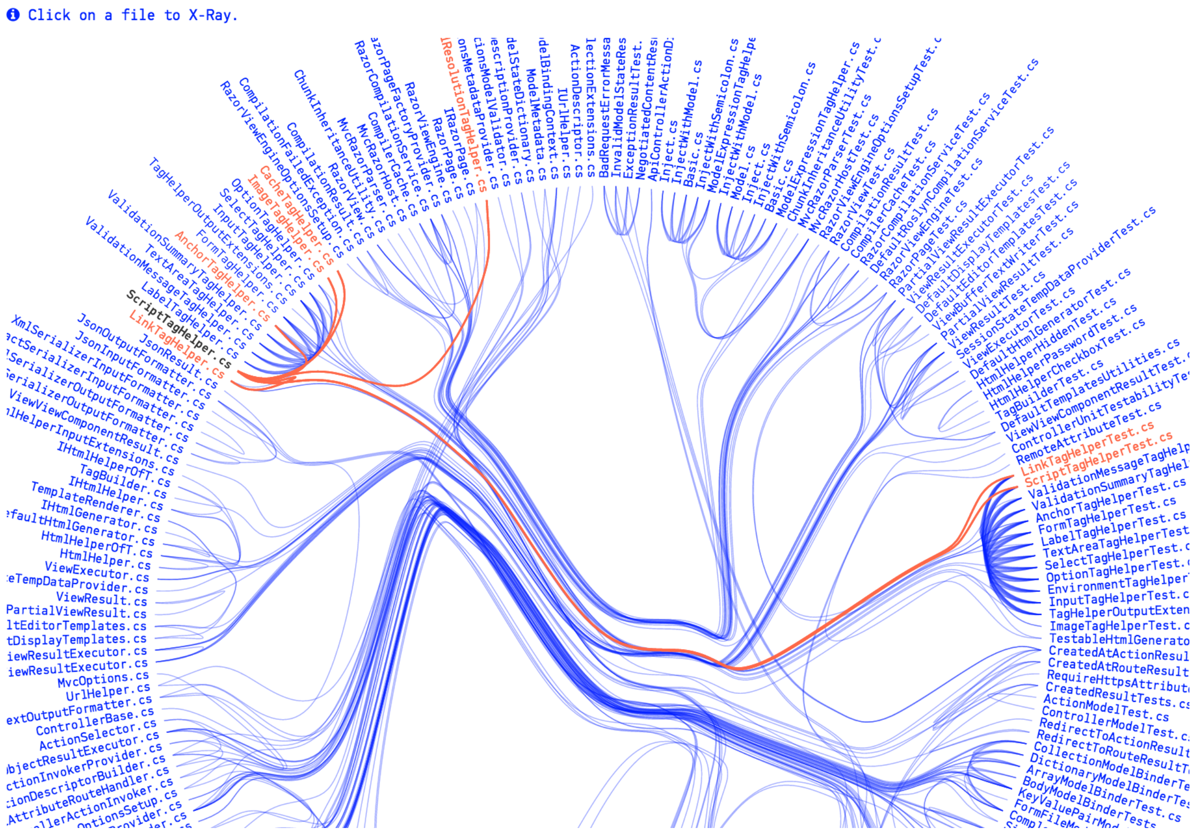
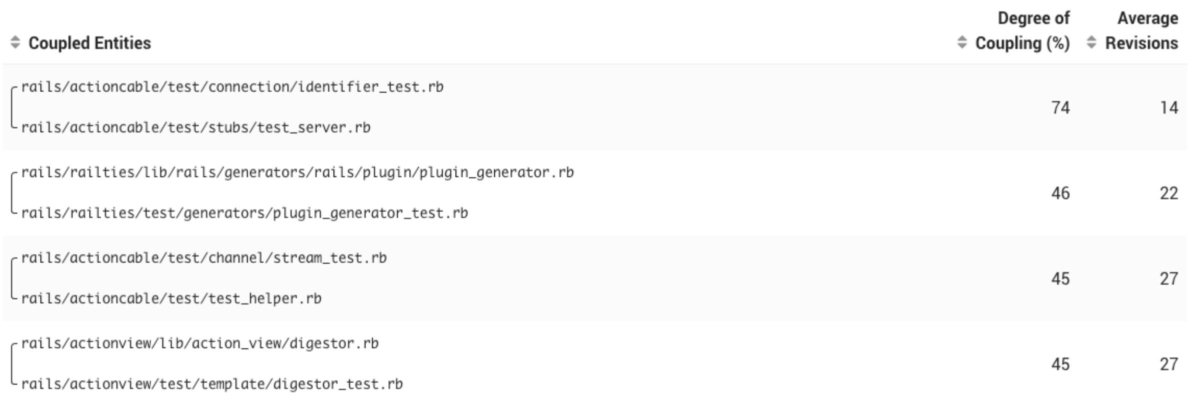


Fig. 2.6: Hover over a file in the temporal coupling graph to see its dependants.



Coupled Entities	Degree of Coupling (%)	Average Revisions
rails/actioncable/test/connection/identifier_test.rb rails/actioncable/test/stubs/test_server.rb	74	14
rails/railties/lib/rails/generators/rails/plugin/plugin_generator.rb rails/railties/test/generators/plugin_generator_test.rb	46	22
rails/actioncable/test/channel/stream_test.rb rails/actioncable/test/test_helper.rb	45	27
rails/actionview/lib/action_view/digestor.rb rails/actionview/test/template/digestor_test.rb	45	27

Fig. 2.7: The temporal coupling table gives you all the details.

- Find hidden dependencies in your codebase.

Explore Your Physical Couples

Why do two source code files change together over time? Well, the most common reason is that they have a dependency between them; one is the client of the other. Fig. 2.8 shows an example of such a case.

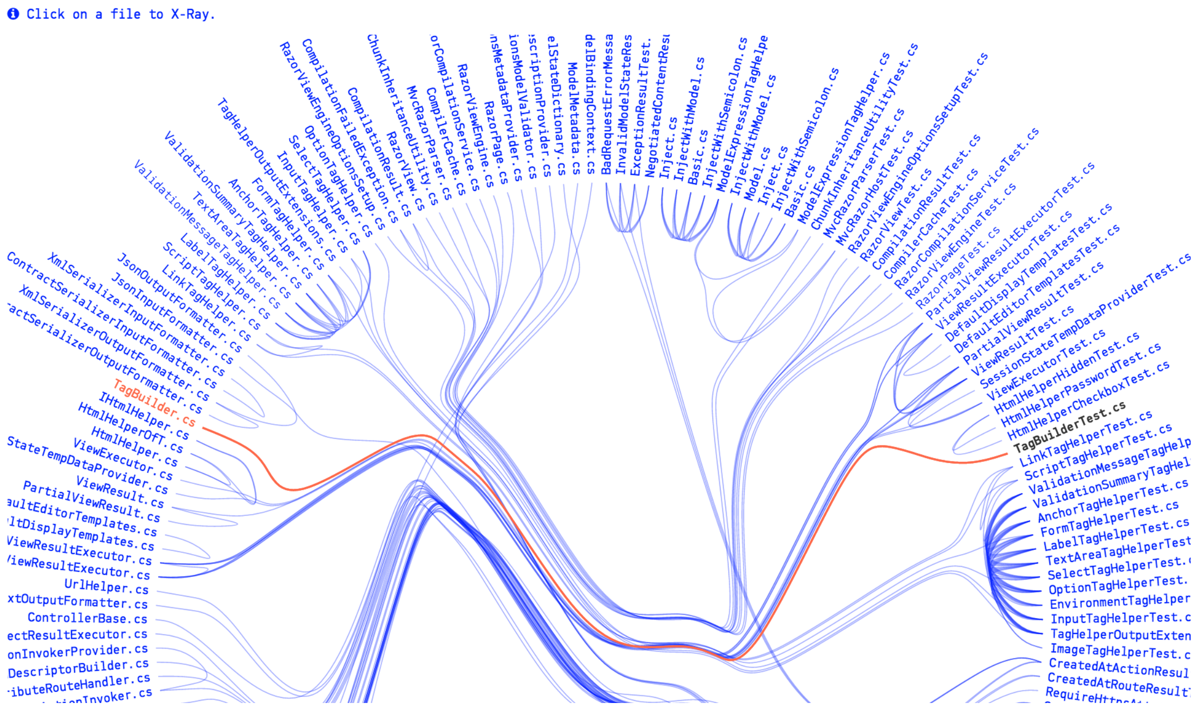


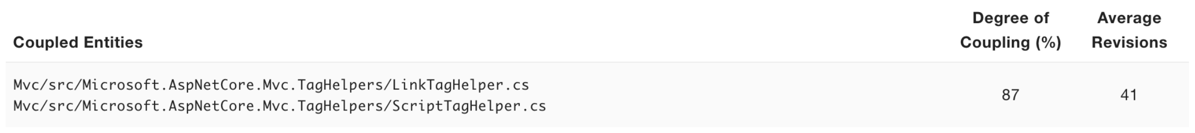
Fig. 2.8: *Temporal sample on unit test.*

As you see in the picture above, a unit test tends to change together with the code under test. This is expected. In fact, we'd be surprised if the temporal coupling was absent - that would be a warning sign since it indicates that your tests aren't being kept up to date or aren't relevant.

A physical dependency like this is something you can detect from the code alone. But remember that Temporal Coupling isn't measured from code; Temporal Coupling is measured from the *evolution* of the code. That means you'll sometimes make unexpected findings.

Look for the Unexpected

Always look for unexpected temporal couples. As soon as you find a logical dependency that you cannot explain, make sure to investigate it. Fig. 2.9 shows an example.



Coupled Entities	Degree of Coupling (%)	Average Revisions
Mvc/src/Microsoft.AspNetCore.Mvc.TagHelpers/LinkTagHelper.cs Mvc/src/Microsoft.AspNetCore.Mvc.TagHelpers/ScriptTagHelper.cs	87	41

Fig. 2.9: *Unexpected temporal coupling.*

The table in Fig. 2.9 shows a strong temporal coupling between a LinkTagHelper.cs and a Script-TagHelper.cs. You also see that their unit tests tend to be changed together.

While those two classes seem to solve related aspects of the same problem, there's no good reason why a change to one of them should imply that the other one has to be changed as well.

When you find an unexpected change pattern like this you need to dig into the code and understand *why*. This is where CodeScene's X-Ray feature proves invaluable (see *X-Ray* (page 26)).

As you X-Ray a temporal coupling cluster you'll often find that there's some duplication of both code and knowledge. Extracting that common knowledge into a module of its own breaks the temporal coupling and makes your code a bit easier to maintain. You see, temporal coupling often suggests refactoring candidates.

Investigate Temporal Dependencies across Architectural Boundaries

Temporal Coupling is like bad weather - it gets worse with the distance you have to travel. In our code, it's a big difference if we need to modify two files located in the same package versus modifying files in different parts of the system. That's why you want to look for temporal dependencies that cross architectural boundaries.

On a side note, some architectures will lead you to exactly those expensive change patterns. The most notable one is a layered architecture. You will often find that most new features implies modifying the majority of your layers. Temporal Coupling helps you keep track of it and assess the situation.

Use Temporal Coupling to predict Omissions

So far you probably got the impression that Temporal Coupling is something to avoid. And you're right. At least in the majority of all cases. But there are some situations where you actually *wants* Temporal Coupling:

- You want your unit tests to evolve with the code under test.
- You want your documentation to be updated together with the system it describes.
- You have parallel implementations for different platforms.

The final point is particular interesting since it shows one of the main strengths of Temporal Coupling: you can identify change patterns *across* different languages and techniques. Fig. 2.10 shows an example from Roslyn, Microsoft's open source compiler platform.

If you have expected Temporal Coupling like this then use it to your advantage. Use the knowledge of your existing development patterns to guide your code reading, commits and to plan your modifications.

Dig Deeper with Sum of Coupling

Sometimes, it may be hard to prioritize if you have a lot of temporal couples in your codebase. In that case, use the *Sum of Coupling* results to guide and prioritize amongst your temporal couples.

Sum of Coupling is a measure of how often a specific file in your codebase is changed together with another file (any other file). The idea is that files that often changes together with others are significant from an architectural perspective.

Change the Temporal Coupling Thresholds depending on your Codebase

In order to avoid biases like large re-organizations of the codebase, CodeScene lets you configure a threshold value for the maximum changeset to consider. This is something you specify in the analysis configuration for your project as illustrated in Fig. 2.11.

The settings in Fig. 2.11 means that the temporal coupling algorithm will respect the following thresholds:

1. Ignore all files with less than 10 revisions/commits since the coupling may be accidental.

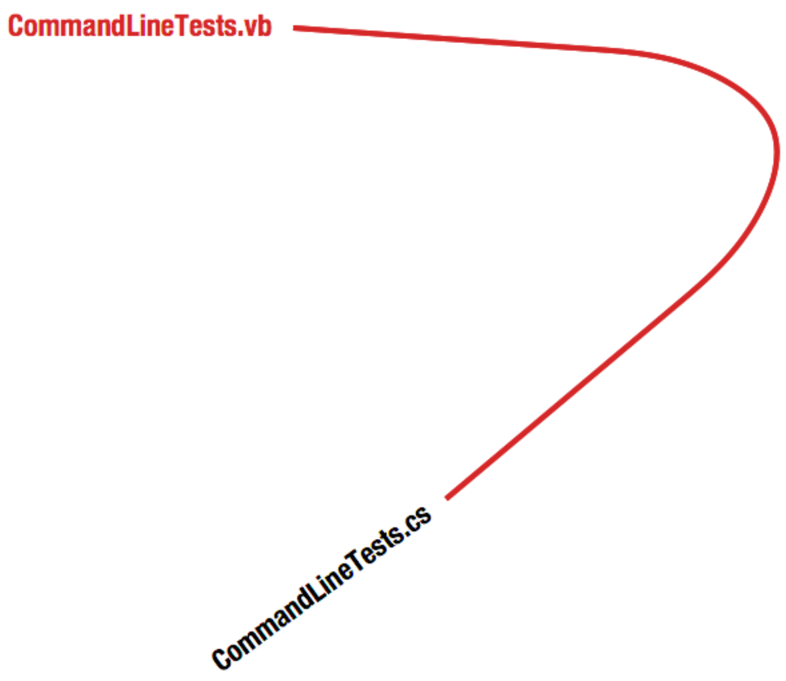
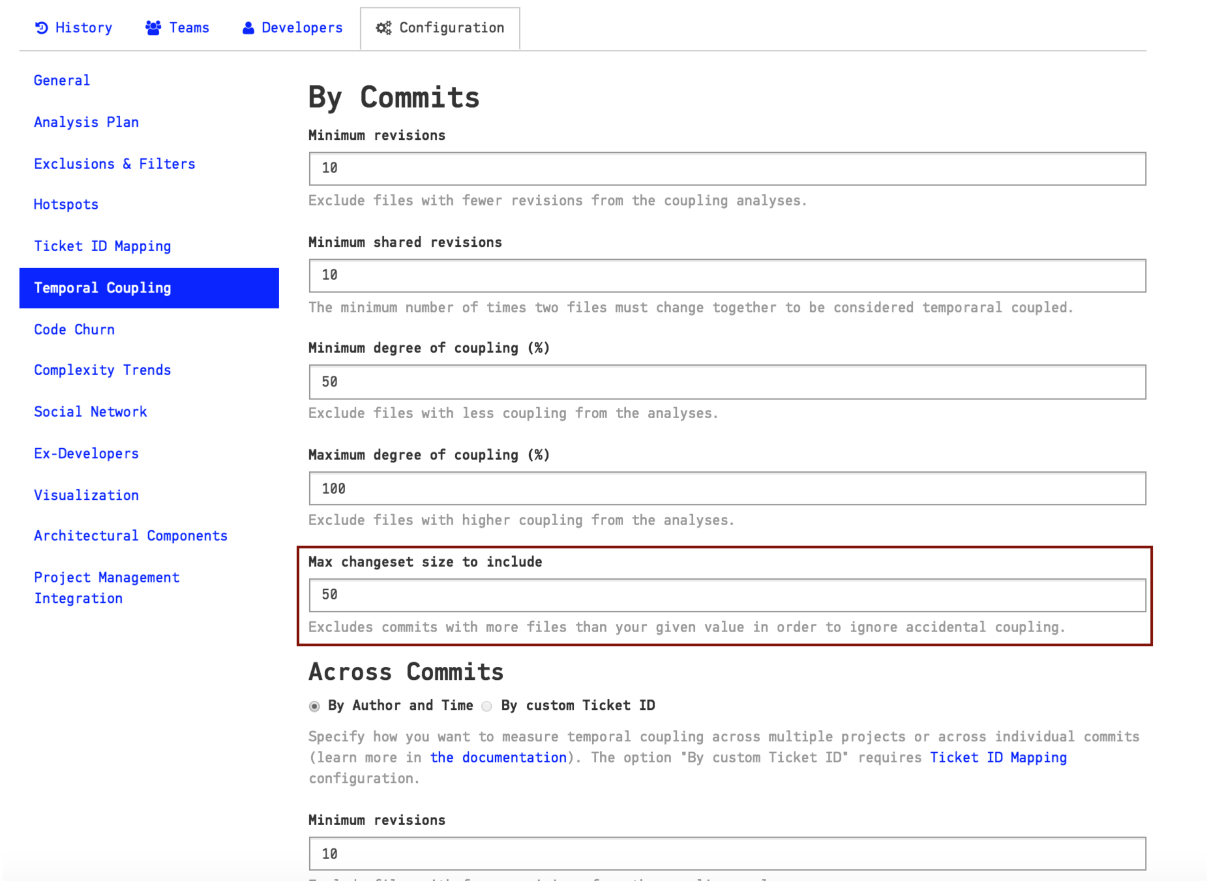


Fig. 2.10: Temporal coupling between languages.



By Commits

Minimum revisions

10

Exclude files with fewer revisions from the coupling analyses.

Minimum shared revisions

10

The minimum number of times two files must change together to be considered temporaral coupled.

Minimum degree of coupling (%)

50

Exclude files with less coupling from the analyses.

Maximum degree of coupling (%)

100

Exclude files with higher coupling from the analyses.

Max changeset size to include

50

Excludes commits with more files than your given value in order to ignore accidental coupling.

Across Commits

[✓ By Author and Time] By custom Ticket ID

Specify how you want to measure temporal coupling across multiple projects or across individual commits (learn more in the documentation). The option "By custom Ticket ID" requires Ticket ID Mapping configuration.

Minimum revisions

10

Fig. 2.11: Temporal coupling configuration.

2. Ignores all temporal couples that haven't co-evolved in at least 10 shared commits since the coupling trend isn't strong enough yet.
3. Ignores all temporal couples with less than 50% strength to filter out the most important coupling.
4. Ignores all changesets/commits where more that 50 files were changed together since we want to limit potential false positives.

You'll find that the default values are typically good enough for you initial analyses. You typically lower the thresholds in case you don't find any temporal coupling. Similarly, you increase the thresholds if you analyze a large codebase and get too much analysis data.

Finally, please note that CodeScene lets you specify the thresholds for "Temporal Coupling By Commits" separate from "Temporal Coupling Across Commits". The rationale is because you typically want to use lower thresholds when identifying patterns across commits in different repositories.

Complement Your Intuition

If you're an experienced developer that has contributed a lot of code to a particular project then you probably have a good feeling for where the most significant Hotspots will show-up. You may still get surprised when you run an analysis, but in general most analysis findings will match your intuitive guess. Temporal Coupling is different. We developers seem to completely lack all kind of intuitive sense when it comes to Temporal Coupling.

A Temporal Coupling analysis often gives us deep and unexpected insights into how well our designs stand the test of time.

2.1.3 Complexity Trends

Complexity Trends are used to get more information around our Hotspots.

Once we've identified a number of Hotspots, we need to understand how they evolve; Are they Hotspots because they get more and more complicated over time? Or is it more a question of minor changes to a stable code structure? Complexity Trends help you answer those questions.

Complexity Trends are calculated from the Evolution of a Hotspot

A Complexity Trend is calculated by fetching each historic version of a Hotspot and calculating the code complexity of those historic versions. The algorithm allows us to plot a trend over time as illustrated in Fig. 2.12.

The picture above shows the complexity trend of single hotspot, starting in mid 2015 and showing its evolution over the next year. It paints a worrisome picture since the complexity has started to grow rapidly.

Worse, as evident by the Complexity/Lines of Code ratio shown in Fig. 2.13, the complexity grows non-linear to the amount of new code, which indicates that the code in the hotspot gets harder and harder to understand. You also see that the accumulation of complexity isn't followed by any increase in descriptive comments. So if you ever needed ammunition to motivate a refactoring, well, it doesn't get more evident than cases like this; This file looks more and more like a true maintenance problem.

We'll soon explain how we measure complexity. But let's cover the most important aspect of Complexity Trends first. Let's understand the kind of patterns we can expect.

Know your Complexity Trend Patterns

When interpreting complexity trends, the absolute numbers are the *least* interesting part. You want to focus on the overall shape and pattern first. Fig. 2.14 illustrates the shapes you're most likely to find in a codebase.

Let's have a more detailed look at what the three typical patterns you see above actually mean.

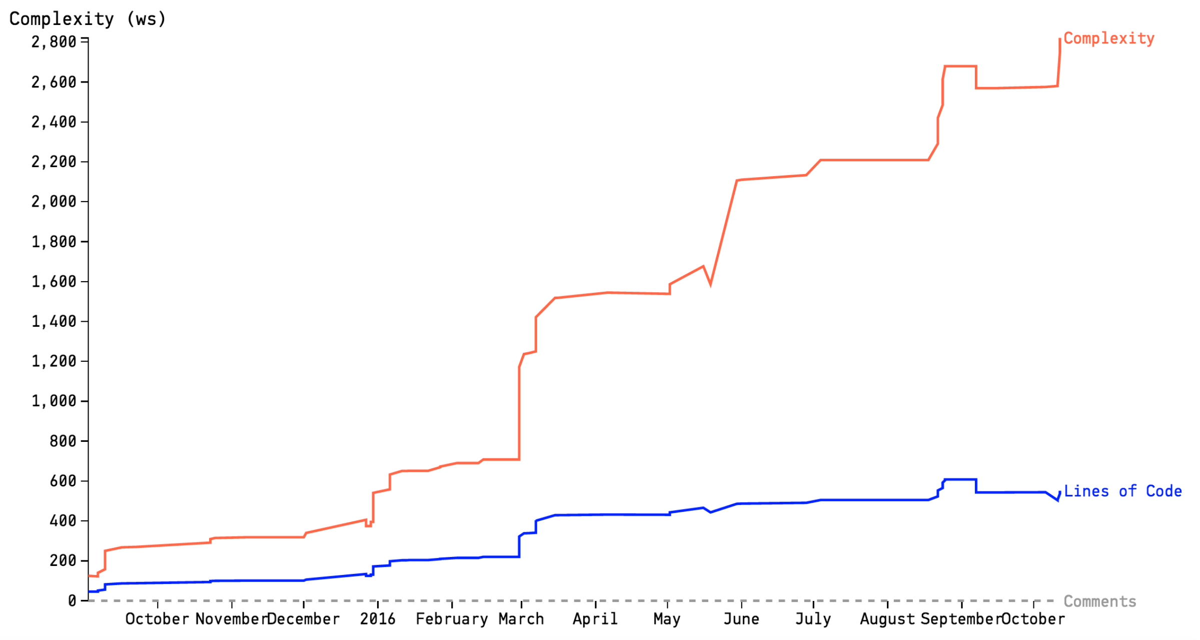


Fig. 2.12: A complexity trend sample.

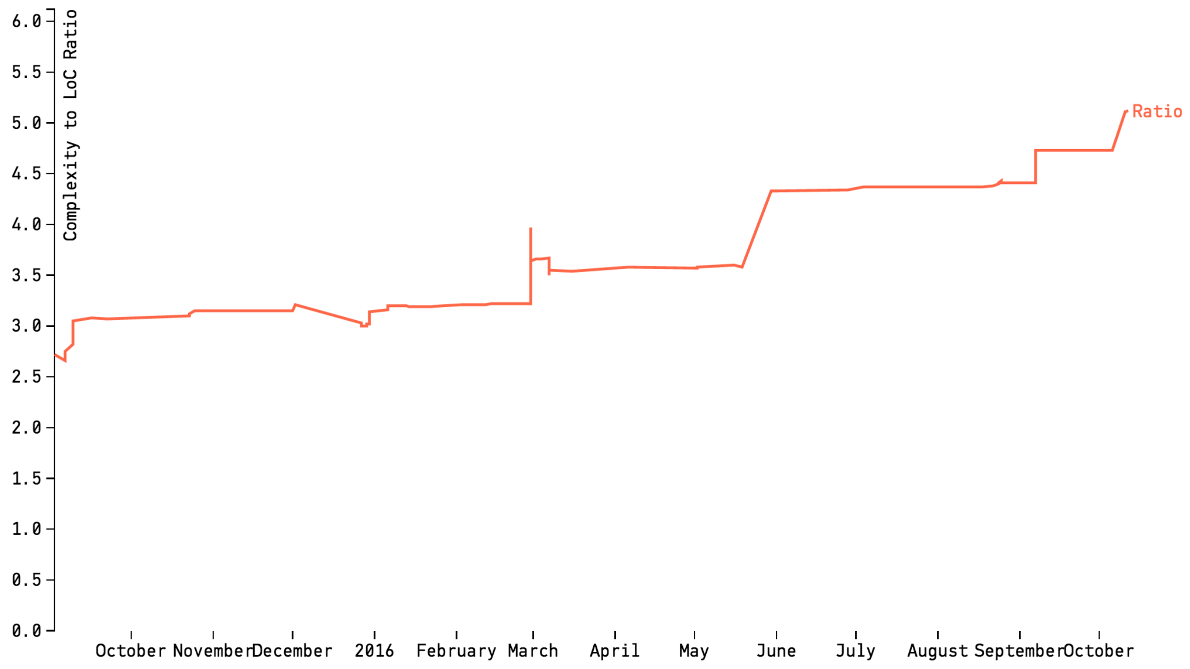


Fig. 2.13: A complexity trend sample.

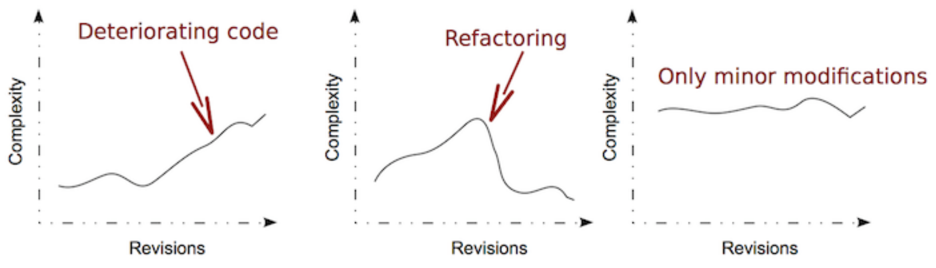


Fig. 2.14: Complexity trend patterns you might find in a codebase.

The Pattern for Deteriorating Code

The pattern to the left, *Deteriorating Code*, is a sign that the Hotspot needs refactoring. The code has kept accumulating complexity. Code does that in either (or both) of the following ways:

1. *Code Accumulates Responsibilities:* A common case is that new features and requirements are squeezed into an existing class or module. Over time, the unit's cohesion drops significantly. The consequence of that for our ability to maintain the code is severe; We will now have to change the same unit of code for many different reasons. Not only does it put us at risk for unexpected feature interactions and defects. It's also harder to re-use the code and to modify it due to the excess cognitive load we face in a module with more or less related functionality.
2. *Constant Modification to a Stable Structure:* Another common reason that code become hotspots is because of a low-quality implementation. We constantly have to re-visit the code, add an if-statement to fix some corner case and perhaps introduce that missing else-branch. Soon, the code becomes a maintenance nightmare of mythical proportions (you know, the kind of code you use to scare new recruits).

Complexity Trends let you detect these two potential problems early. Once you've found them, you need to refactor the code. And Complexity Trends are useful to track your improvements too. Let's see how.

Track Improvements with Complexity Trends

Have one more look at the picture above. You see the second pattern, "Refactoring"? A downward slope in a complexity trend is a good sign. It either means that your code gets simpler (perhaps as those nasty if-else-chains get refactored into a polymorphic solution) or that there's less code because you extract unrelated parts into other modules.

Now, please pat yourself on the back if you have the Refactoring trend in your hotspots - it's great! But do keep exploring the complexity trends; What often happens is that we spend an awful amount of time and money on improving something, fail to address the root cause, and soon the complexity slips back in. Fig. 2.15 illustrates one such scary case.

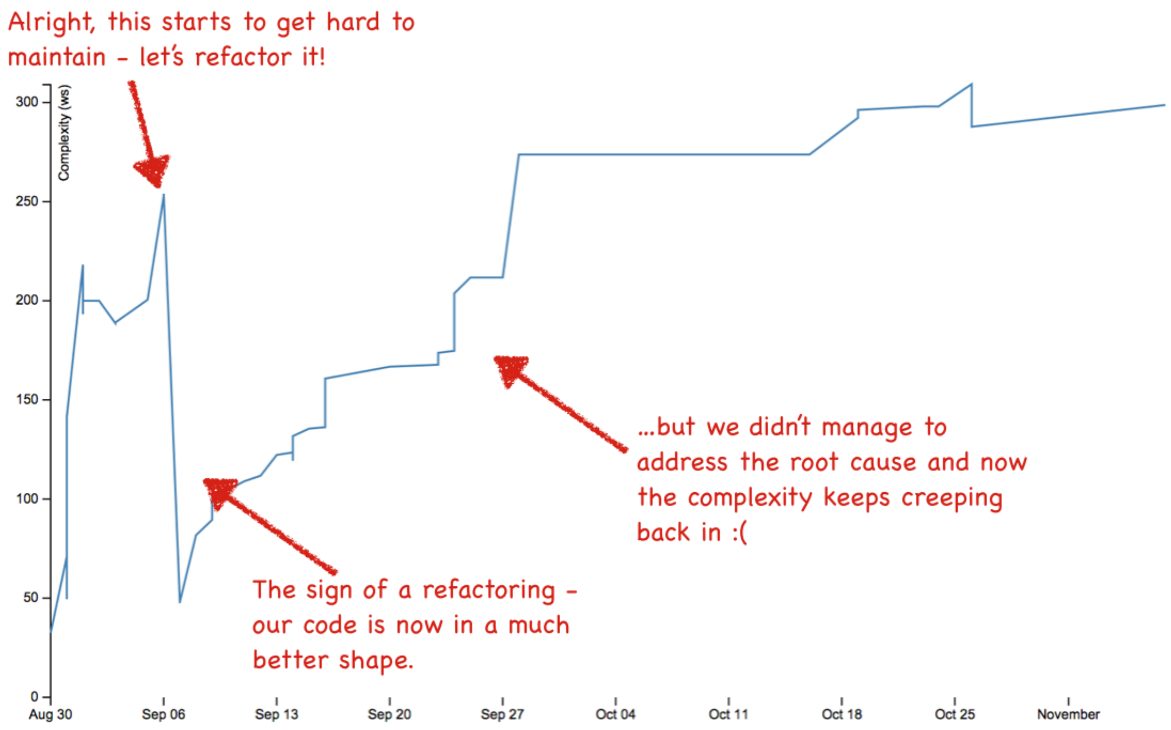


Fig. 2.15: Example on failed refactoring.

You might think this a special case. But let me assure you - during the work on these analysis techniques we analysed hundred of codebases and we found this pattern more often than not. So please, make it a habit to supervise your complexity trend.

Stable Trends May Indicate Problems Too

The final pattern that I want us to discuss is "Only minor modifications". You see an example on that in Fig. 2.16.

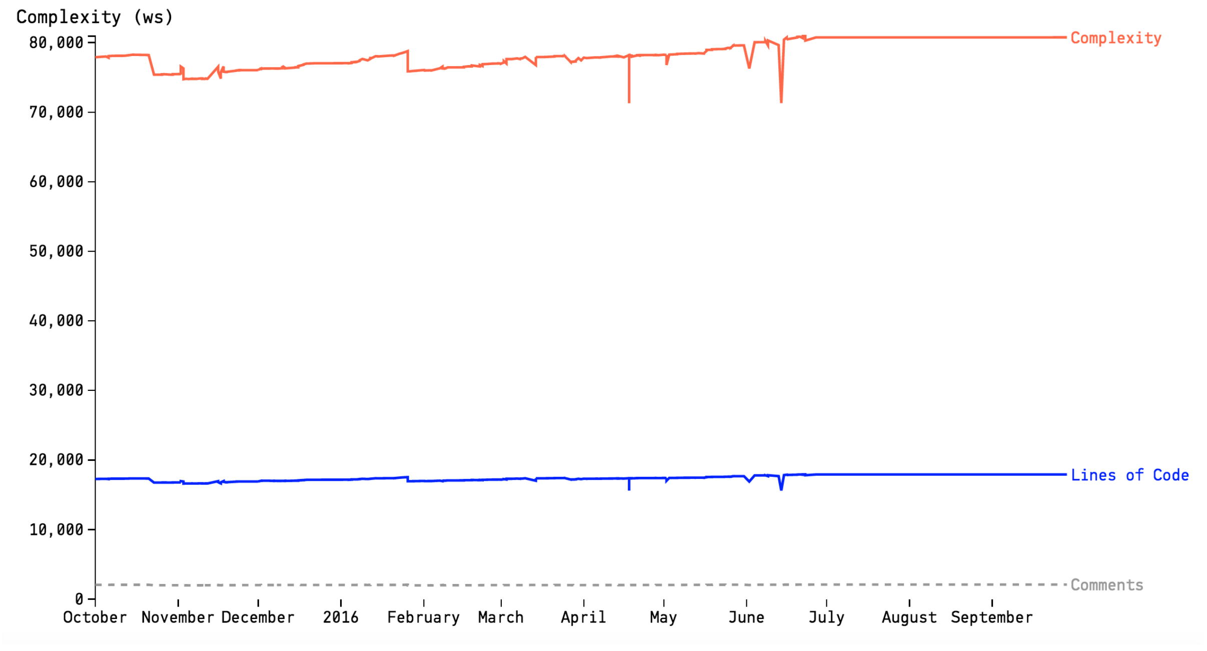


Fig. 2.16: Minor modifications example.

"Minor modifications" - doesn't that sound good? Well, have to to put it into context; The reason you explore a Complexity Trend is because the file is a Hotspot. If a file has become a Hotspot, it means we're making *a lot* of modifications to it. That's a warning sign.

Even if the code itself doesn't become worse of time, as evident by the trend, all those small changes are still bound to be expensive and, potentially, high risk. The key to efficient software maintenance is the *opposite*: you want to stabilize as much of your codebase as you can in terms of development. If a piece of code keeps changing, no matter how small those changes are, it's a sign that either the problem domain isn't well understood or the code fails to model it properly.

Taken together, the "minor modifications" pattern in a Hotspot is a sign that you should try to identify the kinds of modifications you do and try to encapsulate them. This step often involves extracting small, cohesive modules from the Hotspot. The X-Ray feature in CodeScene (see *X-Ray* (page 26)) can help you with that by showing you *when* and *where* you should focus your efforts.

Get Deeper Insights with Descriptive Statistics

The Complexity Trend view presents three additional charts with descriptive statistics:

1. *Max value:* This is the maximum complexity value of a single line of code over time. This graph lets you identify pockets of complexity that need refactoring.
2. *Median value:* The median is the complexity value that separates the higher half of values from the lower half. An increase in median is a sign that your code doesn't just grow in terms of new lines; Rather, it's a sign that the existing code becomes more complicated.

3. *Standard Deviation:* A standard deviation tells you how much the complexity of individual lines vary. The lower the standard variation figure, the more alike the lines of code in your program (probably a good sign).

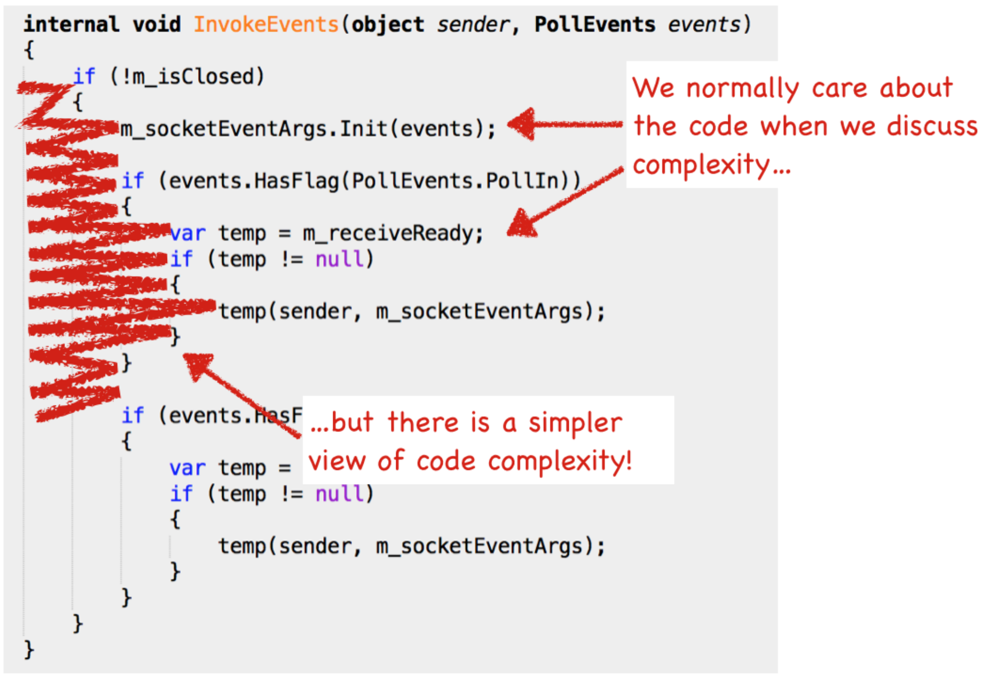
The descriptive statistics require some training to interpret. But as a rule of thumb: if any trend increases, that's a bad sign.

What's "Complexity" anyway?

Alright, we said that Complexity Trends calculate the complexity of historic versions of our Hotspots. So what kind of metric do we use for complexity?

The software industry has several well-known metric. You might have heard about Cyclomatic Complexity or Halstead's volume measurement. These are just two examples. What all complexity metrics have in common, however, are that they are pretty bad at predicting complexity!

So we'll use a less known metric, but one that has been shown to correlate well with the more popular metrics. We'll use *indentation-based complexity* as illustrated in Fig. 2.17



```
internal void InvokeEvents(object sender, PollEvents events)
{
    if (!m_isClosed)
    {
        m_socketEventArgs.Init(events);

        if (events.HasFlag(PollEvents.PollIn))
        {
            var temp = m_receiveReady;
            if (temp != null)
            {
                temp(sender, m_socketEventArgs);
            }
        }

        if (events.HasF
        {
            var temp =
            if (temp != null)
            {
                temp(sender, m_socketEventArgs);
            }
        }
    }
}
```

We normally care about the code when we discuss complexity...
...but there is a simpler view of code complexity!

Fig. 2.17: Explaining whitespace complexity.

Virtually all programming languages use whitespace as indentation to improve readability. In fact, if you look at some code, any code, you'll see that there's a strong correlation between your indentations and the code's branches and loops. Our indentation-based metric calculates the number of indentations (tabs are translated to spaces) with comments and blank lines stripped away.

Indentation-based complexity gives us a number of advantages:

- It's language-neutral, which means you get the same metric for Java, JavaScript, C++, Clojure, etc. This is important in today's polyglot codebases.
- It's fast to calculate, which means you don't have to wait half a day to get your analysis results.

Know the Limitations of Indentation-Based Complexity

Of course, there's no such thing as a perfect complexity metric. Indentation-based complexity has a number of pitfalls and possible biases. Let's discuss them so that you can keep an eye at them as you interpret the trends in your own code:

- *Sensitive to layout changes:* If you change your indentation style midway through a project, you run the risk of getting biased results. In that case you need to know at what date you made that change and use that when interpreting the results.
- *Sensitive to individual differences in style:* Let's face it - you want a consistent style within the same module. Inconsistent indentation styles makes it harder to manually scan the code. So please settle on a shared style.
- *Does not understand complex language constructs:* There are certain language constructs that indentation-based complexity will treat as simple although the opposite may hold true. Examples include list compressions and their relatives like the stream API in Java 8 or LINQ in .NET. On the other hand, it's common to add line breaks and indent those constructs as well.

Alright, we're through this guide on Complexity Trends and you're ready to explore the patterns in your own codebase. Just remember that, like all models of complex processes, Complexity Trends are an heuristic - not an absolute truth. They still need your expertise and knowledge about the codebase's context to interpret them.

2.1.4 X-Ray

X-Ray gives you Deep Insights into your Code

Hotspots are code with high change frequencies. We know that any improvements we do to a hotspot are likely to pay-off immediately. However, sometimes those improvements aren't straightforward; Some of the worst hotspots we've seen are files with several thousands lines of code. Given that amount of code, where do we start? Are all parts of that file equally important? Are there any functions or methods that contribute more to the code being a hotspot than others?

Until recently, this is where the CodeScene analyses stopped. After all, we've significantly reduced the amount of code we need to consider as we narrowed down a whole codebase to a single file where improvements matter. However, we need to do even better and CodeScene's X-Ray feature fills this gap.

X-Ray is a language-dependent analysis. X-Ray is available for the following programming languages:

- C
- C++
- C#
- Java
- Clojure
- Ruby
- Python
- Erlang
- Scala
- JavaScript/ECMAScript

We'll continue to add support to X-Ray for more programming languages over time. As always: if you lack support for a language, let us know and we'll make it happen.

An Overview of X-Ray

X-Ray is an analysis that operates on the function/method level of your code. Thus, X-Ray is able to provide deep and detailed information on what's happening *inside* a Hotspot.

There are three main use cases for the X-Ray functionality:

1. X-Ray lets you make sense of large files and get specific recommendations on the parts to improve.
2. X-Ray provides detailed information on why a cluster of files are temporally coupled.
3. X-Ray recommends re-structuring opportunities on the methods in your Hotspots in order to make the code easier to understand and maintain.

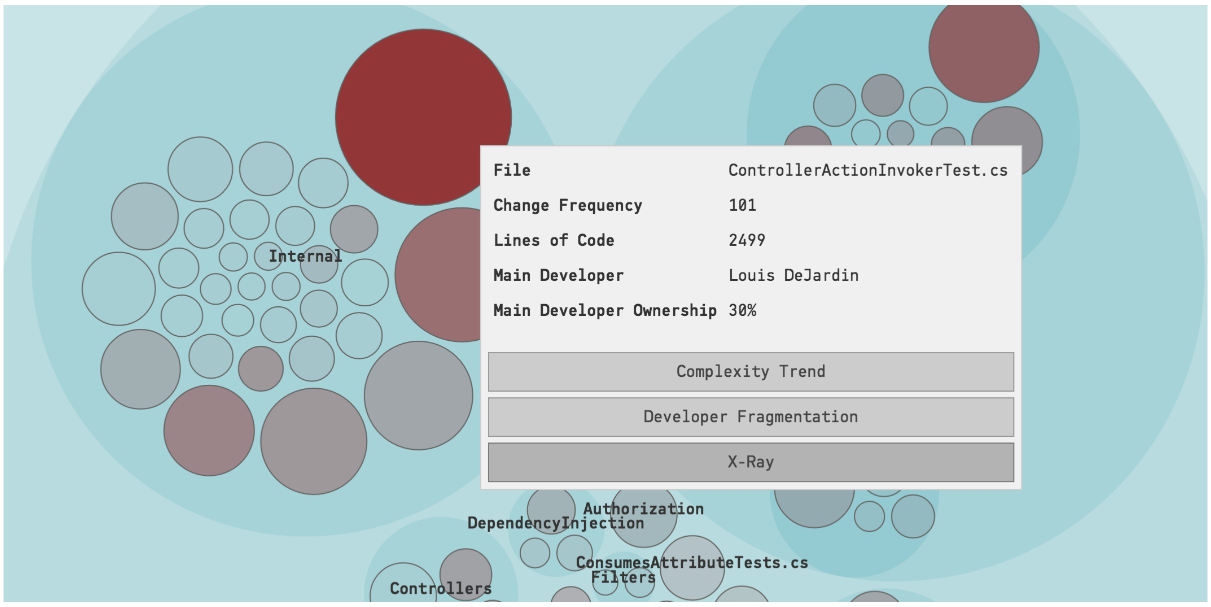
In the following guide we'll cover all of these cases. Let's start with how you can make sense of large files.

X-Ray calculates Hotspots on a Method Level

A Hotspot analysis is orthogonal to the data it operates on. That is, CodeScene presents hotspots as individual files, but also on an architectural level as entire components and sub-systems. With X-Ray, we climb down the abstraction ladder and run a Hotspot analysis on a method level.

A large file is like a system in itself. Some parts remain stable, while other parts of the file keeping changing as new features are added and bugs get resolved. With X-Ray, you'll get a prioritized list of the methods you want to refactor and improve first. This is important since re-designing a large module is both high-risk and expensive. So instead you want to take an iterative approach to your improvements and base those improvements on data.

To run X-Ray, go to your Hotspot map, click on the Hotspot and select 'X-Ray' from the context menu as shown in Fig. 2.18.



File	ControllerActionInvokerTest.cs
Change Frequency	101
Lines of Code	2499
Main Developer	Louis DeJardin
Main Developer Ownership	30%

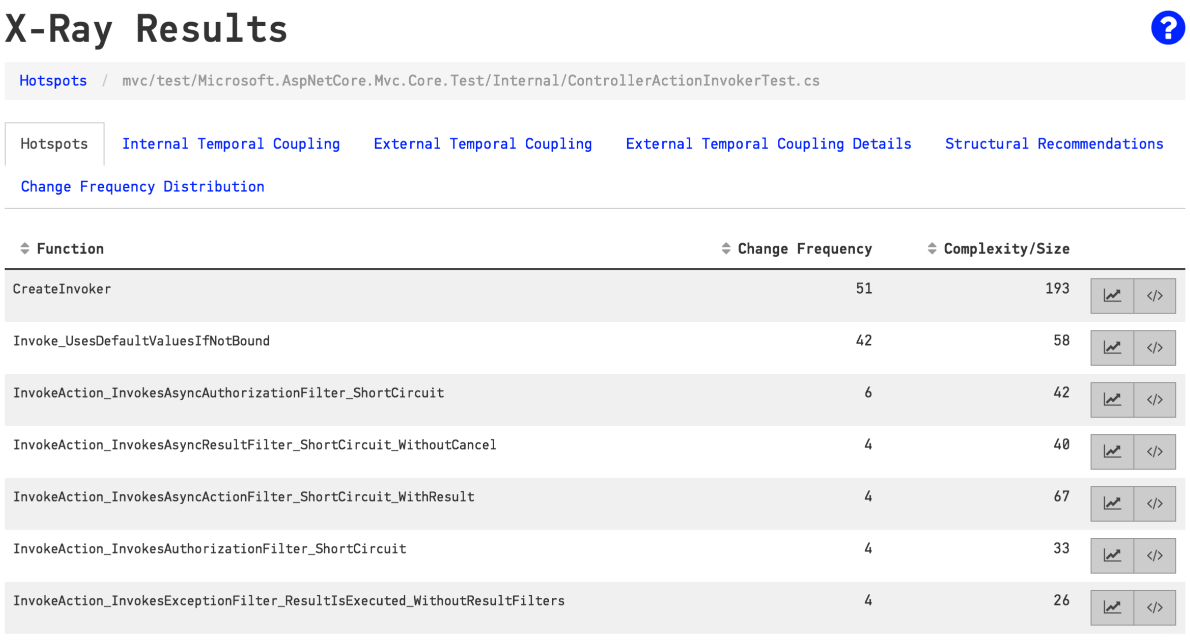
Fig. 2.18: Run X-Ray from the context menu.

X-Ray is run on demand. That is, the first time you execute it on a Hotspot it may take a few seconds to get the results. Sub-sequent accesses are cheap since we cache the results.

Once you get the results you'll see that you typically spend more time on some methods than others. So let's walk through the X-Ray results and look at the individual pieces. Have a look Fig. 2.19 as a starting-point.

Fig. 2.19 shows the results of an X-Ray analysis. We see that our hotspot is a method named *CreateInvoker*, which consists of 193 lines of code. This method is where you'd like to focus your refactoring efforts; The high change frequency of the method indicates that improvements are likely to pay-off immediately.

You also see that each row in the table above lets you run a Complexity Trend analysis. In this case, the trend analysis will show the complexity growth of an individual method. Look at the results of those



X-Ray Results

Hotspots / mvc/test/Microsoft.AspNetCore.Mvc.Core.Test/Internal/ControllerActionInvokerTest.cs

Hotspots | Internal Temporal Coupling | External Temporal Coupling | External Temporal Coupling Details | Structural Recommendations | Change Frequency Distribution

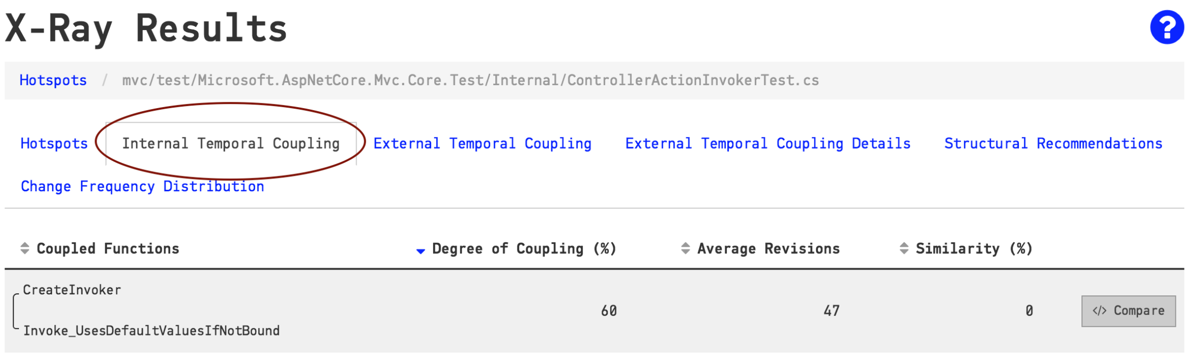
Function	Change Frequency	Complexity/Size
CreateInvoker	51	193
Invoke_UsesDefaultValuesIfNotBound	42	58
InvokeAction_InvokesAsyncAuthorizationFilter_ShortCircuit	6	42
InvokeAction_InvokesAsyncResultFilter_ShortCircuit_WithoutCancel	4	40
InvokeAction_InvokesAsyncActionFilter_ShortCircuit_WithResult	4	67
InvokeAction_InvokesAuthorizationFilter_ShortCircuit	4	33
InvokeAction_InvokesExceptionFilter_ResultIsExecuted_WithoutResultFilters	4	26

Fig. 2.19: The starting point in an X-Ray analysis.

trends to determine if the X-Ray hotspot represents a method that we've already started to refactor or, the more common case, represents code that continues to degrade in quality.

X-Ray calculates Temporal Coupling between Methods

As you X-Ray a Hotspot, CodeScene also looks for temporal coupling *between* individual methods in that file. This is information that helps you identify unexpected change patterns. Let's look the example in Fig. 2.20.



X-Ray Results

Hotspots / mvc/test/Microsoft.AspNetCore.Mvc.Core.Test/Internal/ControllerActionInvokerTest.cs

Hotspots | Internal Temporal Coupling | External Temporal Coupling | External Temporal Coupling Details | Structural Recommendations | Change Frequency Distribution

Coupled Functions	Degree of Coupling (%)	Average Revisions	Similarity (%)
CreateInvoker Invoke_UsesDefaultValuesIfNotBound	60	47	0

Fig. 2.20: X-Ray calculates temporal coupling between the methods in your Hotspot.

Fig. 2.20 shows that two methods, *CreateInvoker* and *Invoke_UsesDefaultValuesIfNotBound* changes together in 60% of all changes. That is, every second time you change one of these methods there's a predictable change to the other one.

You use the Temporal Coupling results as input to your refactoring efforts. For example, in the example above, you probably want to have a close look at both methods to see why they are so strongly coupled in time. Often, there's either a leaky abstraction or a fair chunk of duplicated logic in either part of the code.

X-Ray lets you look into Temporal Coupling Clusters

Temporal Coupling is one of the most powerful software analyses in our arsenal. A temporal coupling analysis often highlights unexpected change patterns in our codebase and provides us with important information that we cannot deduce from the code alone. However, temporal coupling has also been one of the hardest results to act upon.

Think about it for a minute. Let's say that you investigate some temporal coupling results and identify a cluster of 10 files that tend to change together. Now, how do you uncover the reason for this coupling in time? Well, in more complex cases you need to compare the code and walk through the historic revisions to know which parts of the files that are responsible for the coupling. This can be painful, particularly for large files that are low on cohesion. Enter X-Ray for temporal coupling.

With X-Ray, all of these steps are completely automated. You just click on a file in the temporal coupling visualization and select 'X-Ray' from the context menu as illustrated in Fig. 2.21.

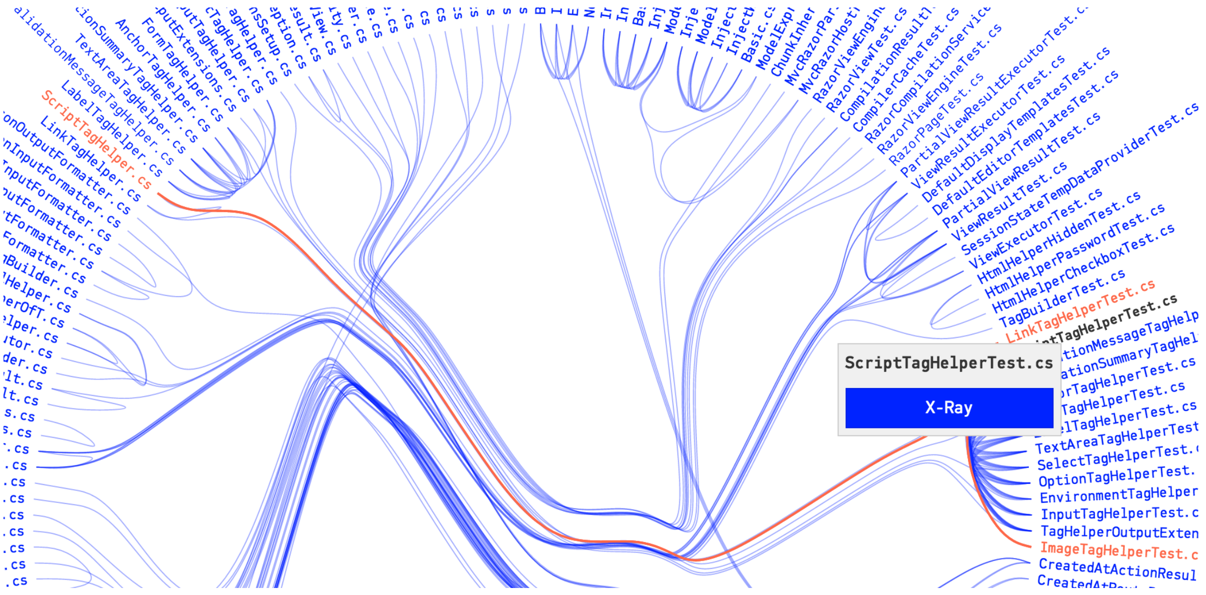


Fig. 2.21: X-Ray lets you investigate temporal coupling clusters in detail.

Once X-Ray is done, you're presented with a dependency wheel on method level. Have a look the dependency wheel in Fig. 2.22 and I'll walk you though the details.

The dependency wheel in Fig. 2.22 is an interactive visualization. As you see in the example above, when we hover over the part that represents the method *RendersLinkTagsForGlobbedHrefResults*, we see that the method is coupled in time to six other methods located in a different class. This information is powerful: now we've limited the amount of code you need to inspect in order to improve the design and break this expensive change pattern.

X-Ray detects Software Clones

Temporal coupling arises for several reasons. It's also important to note that all coupling isn't bad. For example, you'd expect a unit test to change together with the code under test. However, in the case where you can't think about any good reason two pieces of code keep changing at the same time you'll inevitably find a refactoring opportunity.

One of the most common reasons for unexpected temporal coupling is a dear old friend: copy-paste. In fact, copy-paste is so common that we've included an analysis of code similarity in X-Ray.

You get to the code similarity analysis by clicking at the result tab for External Temporal Coupling Details as illustrated in Fig. 2.23.

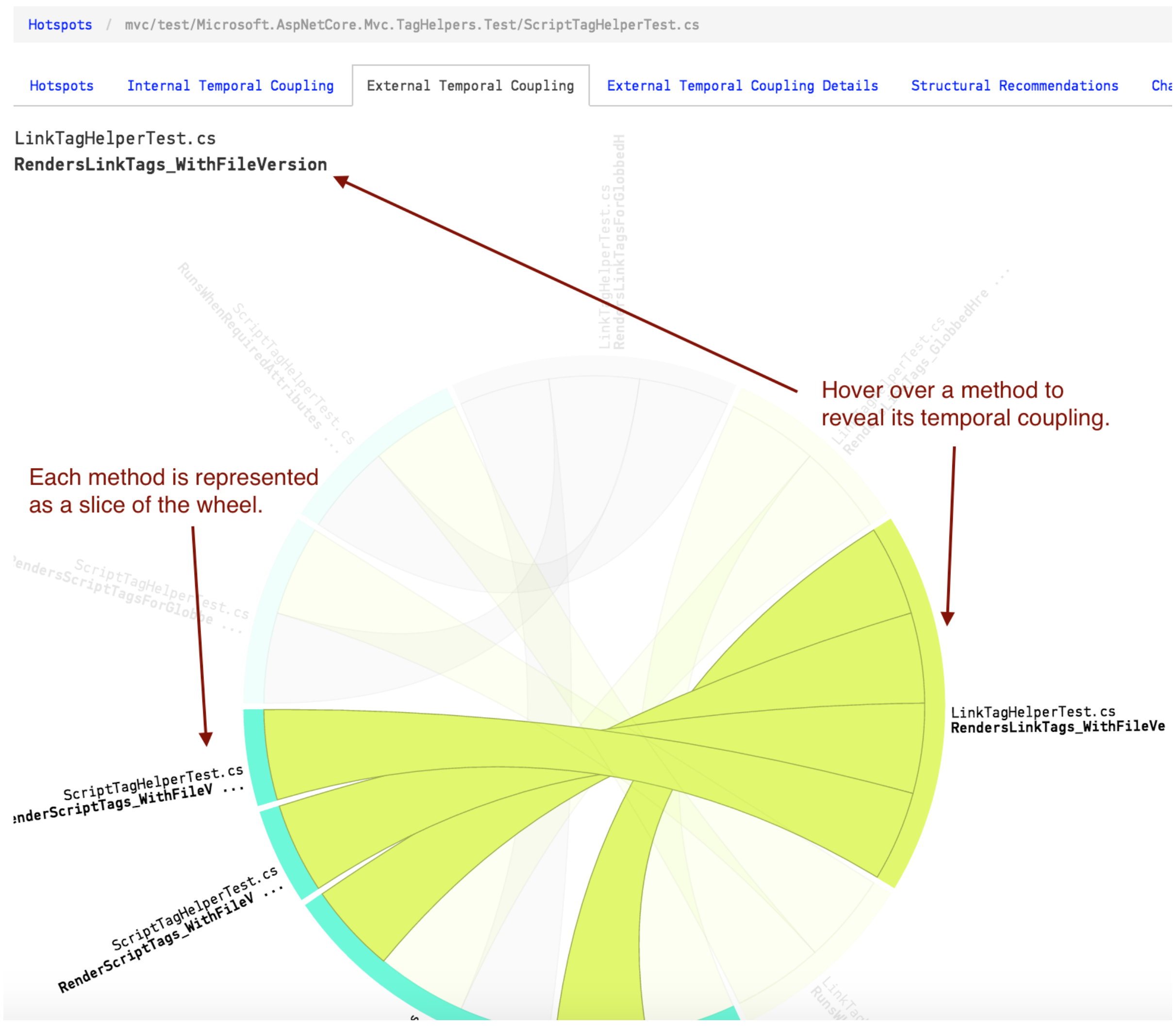
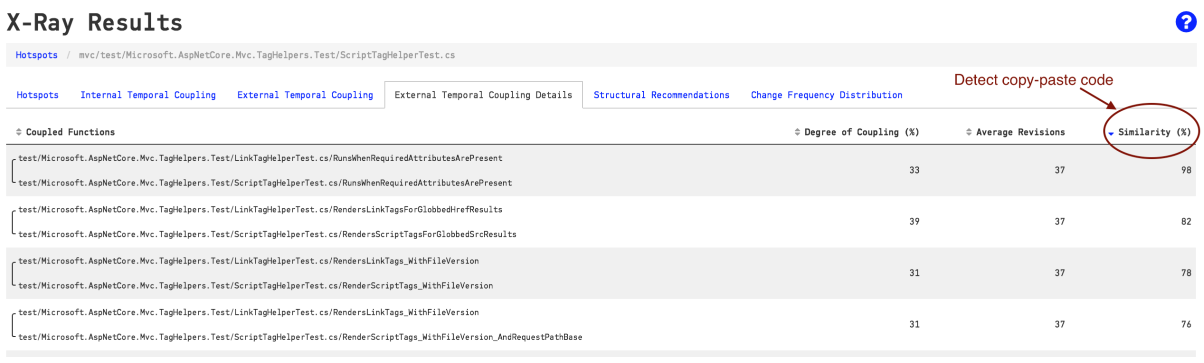

Hotspots / mvc/test/Microsoft.AspNetCore.Mvc.TagHelpers.Test/ScriptTagHelperTest.cs
Hotspots
Internal Temporal Coupling
External Temporal Coupling
External Temporal Coupling Details
Structural Recommendations
LinkTagHelperTest.cs
RendersLinkTags_WithFileVersion
Hover over a method to reveal its temporal coupling.
Each method is represented as a slice of the wheel.
LinkTagHelperTest.cs
RendersLinkTags_WithFileVe
ScriptTagHelperTest.cs
RenderScriptTags_WithFileV ...
ScriptTagHelperTest.cs
RenderScriptTags_WithFileV ...

Fig. 2.22: The dependency wheel shows the temporal coupling between methods.


Hotspots
Internal Temporal Coupling
External Temporal Coupling
External Temporal Coupling Details
Structural Recommendations
Change Frequency Distribution
Detect copy-paste code

X-Ray Results

Hotspots / mvc/test/Microsoft.AspNetCore.Mvc.TagHelpers.Test/ScriptTagHelperTest.cs

Coupled Functions	Degree of Coupling (%)	Average Revisions	Similarity (%)
test/Microsoft.AspNetCore.Mvc.TagHelpers.Test/LinkTagHelperTest.cs/RunsWhenRequiredAttributesArePresent test/Microsoft.AspNetCore.Mvc.TagHelpers.Test/ScriptTagHelperTest.cs/RunsWhenRequiredAttributesArePresent	33	37	98
test/Microsoft.AspNetCore.Mvc.TagHelpers.Test/LinkTagHelperTest.cs/RendersLinkTagsForGlobbedHrefResults test/Microsoft.AspNetCore.Mvc.TagHelpers.Test/ScriptTagHelperTest.cs/RendersScriptTagsForGlobbedSrcResults	39	37	82
test/Microsoft.AspNetCore.Mvc.TagHelpers.Test/LinkTagHelperTest.cs/RendersLinkTags_WithFileVersion test/Microsoft.AspNetCore.Mvc.TagHelpers.Test/ScriptTagHelperTest.cs/RenderScriptTags_WithFileVersion	31	37	78
test/Microsoft.AspNetCore.Mvc.TagHelpers.Test/LinkTagHelperTest.cs/RendersLinkTags_WithFileVersion test/Microsoft.AspNetCore.Mvc.TagHelpers.Test/ScriptTagHelperTest.cs/RenderScriptTags_WithFileVersion_AndRequestPathBase	31	37	76

Fig. 2.23: The Code Similarity analysis let you uncover copy-paste code.

In Fig. 2.23 you see that there are two methods with the same name, but located in different classes, that have a code similarity of 98%. You want to use this data as a starting point. If you could encapsulate that shared logic in a separate method that you re-use between the two classes your temporal coupling will go away. Your application will become a little bit easier to maintain.

A word on Software Clone Detection

Copy-paste detection isn't exactly a new technique. However, it's still far from mainstream in the software industry. One reason that copy-paste detectors haven't caught on is because they fail to prioritize their findings in a sensible way.

If you look at studies of large codebases, you'll learn that around 5-20% of all large codebases represents duplicated logic to some degree. That's quite a lot. There's simply no way you can start to refactor that amount of code and hope to get a return on that investment. In fact, most of that duplicated code doesn't matter. So how can we find the software clones that limit out ability to maintain the system?

CodeScene's X-Ray solves this dilemma. By combining copy-paste detection with temporal coupling we *know* that the identified software clones matter. For example, if you look at the example above, you'll see that the two methods with a code similarity of 98% are changed together in one third of all cases. That is, with X-Ray you'll find the software clones that actually matter. This lets you prioritize the improvements that you do while still ensuring that you get a real return on those refactoring investments.

Follow the Restructuring Recommendations

Empear's CodeScene is the first ever software analysis tool that implements a *proximity analysis*. The X-Ray findings present the proximity results as a set of recommendations on how to re-structure the methods in a Hotspot in order to make the code more readable. Let's start by understanding the concept of proximity and why it matters to our ability to maintain code.

The proximity principle focuses on how well organized your code is with respect to readability and change. You use proximity both as a design principle and as a heuristic to evaluate the cohesion and structure of existing code.

The principle of proximity is a concept from Gestalt psychology. The Gestalt movement pioneered principles on how we make sense of all chaotic input from our sensory systems. We need to understand the Gestalt principles if we want to optimize our code for readability. Remember, we use the same brain to interpret code as we use to make sense of the physical world.

Within Gestalt psychology, the principle of proximity specifies that objects or shapes that are close to one another appear to form groups as illustrated in Fig. 2.24. If we translate this to software, it means that readable code is structured in a way that lets our brain understand parts of the source code file as a whole. The main reason is because we want our code to support our change patterns: code that is expected to be changed together should be close. Such a code structure serves as a powerful reminder to both the programmer and, more important, the code reader that a set of functions belong together.

CodeScene measures proximity based on your change patterns (aka internal temporal coupling). You see an example on a proximity analysis in Fig. 2.25 from the implementation of the Clojure programming language.

The highlighted recommendation in Fig. 2.25 shows two functions, *hash-map* and *array-map*, that are frequently changed together. That is, they are temporally coupled. However, if you look at the implementation in the Clojure project you'll see that there are thousands of lines of code between *hash-map* and *array-map*. This is bad news for a maintenance programmer because it's so easy to miss an update to one of the functions. A simple, low-risk refactoring is to just move those two functions next to each other. That simple change lets the code signal that the functions belong together. In addition it dramatically increases the chances that a bug fix to one of the functions is applied to the other function too.

So what metric do we use for proximity? If you look at Fig. 2.25 you see that there's a *Total Proximity* column in the analysis results. The proximity values specify the distance between the related functions. The unit of measure is the number of intermediate functions between the related parts. In our example with *hash-map* and *array-map* Fig. 2.25 shows that there's a total proximity of *299*. That means that

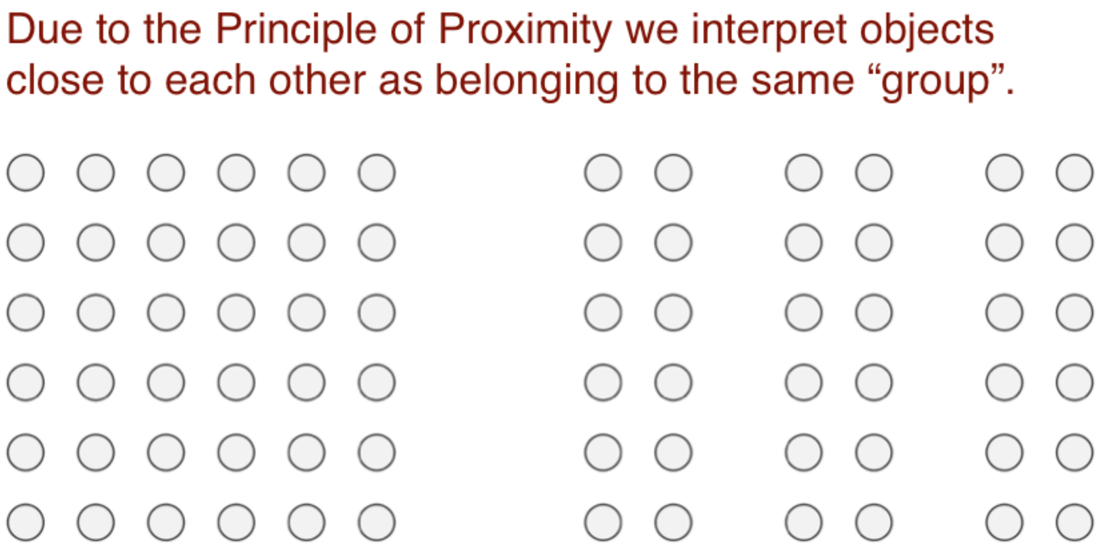
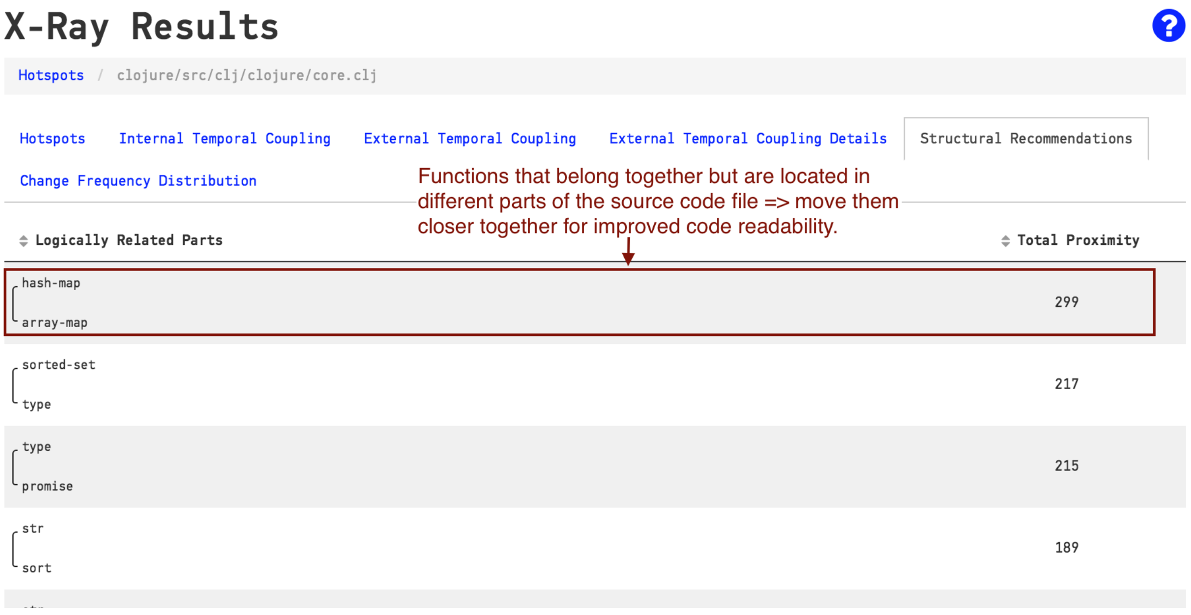

Due to the Principle of Proximity we interpret objects close to each other as belonging to the same "group".

Fig. 2.24: An illustration of the Principle of Proximity where our brain forms groups of related objects.


X-Ray Results
Hotspots / clojure/src/clj/clojure/core.clj
Hotspots
Internal Temporal Coupling
External Temporal Coupling
External Temporal Coupling Details
Structural Recommendations
Change Frequency Distribution
Functions that belong together but are located in different parts of the source code file => move them closer together for improved code readability.

Logically Related Parts	Total Proximity
hash-map / array-map	299
sorted-set / type	217
type / promise	215
str / sort	189

Fig. 2.25: The Proximity Analysis recommends re-structuring of the methods in a Hotspot.

there are 299 (!) functions separating the implementation of *hash-map* from its related temporally coupled *array-map*.

Know the limitations of Method-level analyses

CodeScene tracks renamed content. That is, if you move or rename a file, we make sure to fetch its past history even if you've renamed the file multiple times. We implement a similar mechanism for X-Ray too. X-Ray will track and analyze the history of renamed methods/functions...except when it won't. Let's elaborate on that so that you know the possible corner cases.

First of all we have a philosophical question here. Let's say you decide to refactor parts of your code. You simplify some parts of it and rename a few functions. Now, when is a function renamed and when is it actually a new function that replaces an old one? This distinction isn't clear.

X-Ray resolves this dilemma by introducing a set of heuristics for its rename detection:

1. We consider a method/function renamed if its name is changed *without* any changes to the method body.
2. We also consider a method renamed if its name is changed and their are minor modifications to its method body.
3. X-Ray doesn't do rename detection for methods that it considers too small (e.g. single line getters/setters).

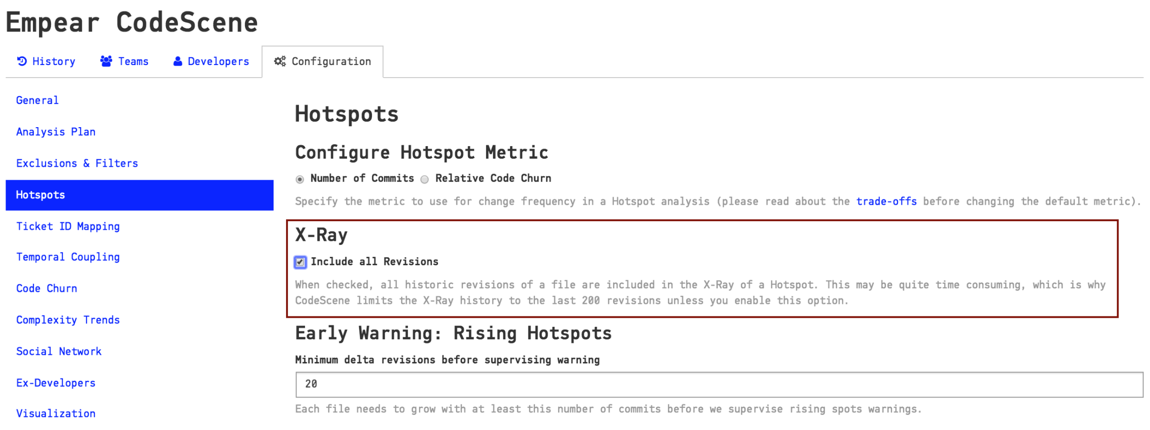
So if you want to ensure that your renamed methods are being tracked past the rename, please make sure that you do the renaming in one commit and possible method body modifications in another commit. It's usually a good refactoring practice anyway.

In general, X-Ray tries to do the most sensible thing. Without the rules above, you'd risk false positives in your analysis results. That's prevented now at the possible cost that X-Ray will miss the occasional rename. This is a better trade-off since if the renamed function is a Hotspot, it will most likely continue to change at a rapid rate and X-Ray detects that anyway.

Increase the Depth of the Analysis

By default, X-Ray will look at a maximum of 200 revisions. In most codebases that's more than enough. So why put a limit on it? Well, there are projects that have been around for a long time and their top Hotspots may well have over thousands of commits. To X-Ray that data will take quite some time. In addition, the most interesting patterns are likely to be in the recent evolution of the Hotspot.

Most of the time this is the behavior that you want. However, in case you want to dive deeper and X-Ray the complete evolution of a Hotspot you need to instruct CodeScene to do that. This choice is a simple matter of configuration as illustrated in Fig. 2.26.



Hotspots

Configure Hotspot Metric

[✓ Number of Commits] Relative Code Churn

Specify the metric to use for change frequency in a Hotspot analysis (please read about the trade-offs before changing the default metric).

X-Ray

[✓ Include all Revisions]

When checked, all historic revisions of a file are included in the X-Ray of a Hotspot. This may be quite time consuming, which is why CodeScene limits the X-Ray history to the last 200 revisions unless you enable this option.

Early Warning: Rising Hotspots

Minimum delta revisions before supervising warning

20

Each file needs to grow with at least this number of commits before we supervise rising spots warnings.

Fig. 2.26: The project configuration lets you X-Ray all revisions of a Hotspot.

2.1.5 Code Churn

Code churn is a measure that tells you the rate at which your code evolves. Code churn has several usages:

- *Visualize your development process:* Your code churn signature in the diagrams below mirrors the practices you use to deliver code. You may want to watch out for regular spikes, which may hint at a mini waterfall going on in your daily work
- *Reason about delivery risks:* Code churn is a good predictor of post-release defects. Thus, it's a warning sign if you approach a deadline while your code churn increases. That's a sign that the code gets more and more volatile the closer you get to your deadline. You want the opposite. You want to stabilize more and more code the closer you get to delivery.

CodeScene provides several churn measures. They're all described in this guide and you typically investigate all of them to get the overall trend in your codebase.

Use the Commit Activity as the Pulse of your Codebase

The commit activity chart shows the number of commits and contributing authors over time as illustrated in Fig. 2.27.

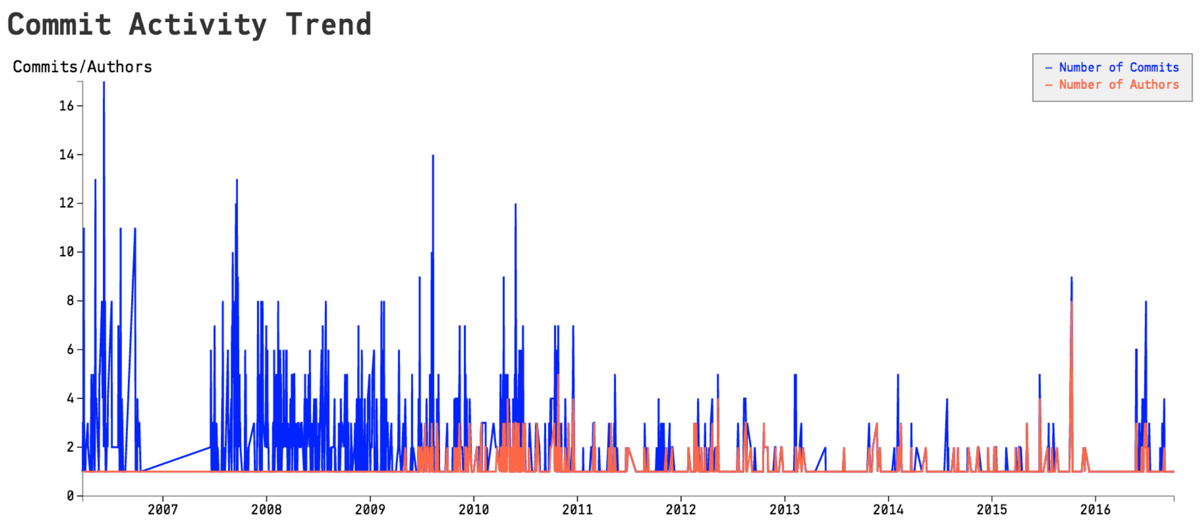


Fig. 2.27: The commit activity chart.

The number of commits and authors over time is a different kind of churn. This metric will typically correlate well with your other Code Churn metrics described below. However, you want to look out for potential productivity issues like an increase in authors without a corresponding increase in commits and churn; Such a trend often indicates that you've more programmers contributing than the software architecture (and/or organization) can support.

Uncover Long-term Trends in your Code Churn Rolling Average

CodeScene measures two separate churn metrics: the number of added lines of code, and the number of deleted lines. The values in the graph shows the rolling average of your code churn (rolling average is a technique to smooth out sudden fluctuations in your data). You configure a time window for the rolling average in your project configuration.

The main use of these code churn metrics is to reason about delivery risks; If you're close to a deadline and have a rising churn, you might want to understand *why*. After all, an increase in churn means that the codebase has become more and more volatile. Unless you have an extensive safety net in terms of tests and continuous deployment techniques, you probably want to stabilize before a release as illustrated in Fig. 2.28.

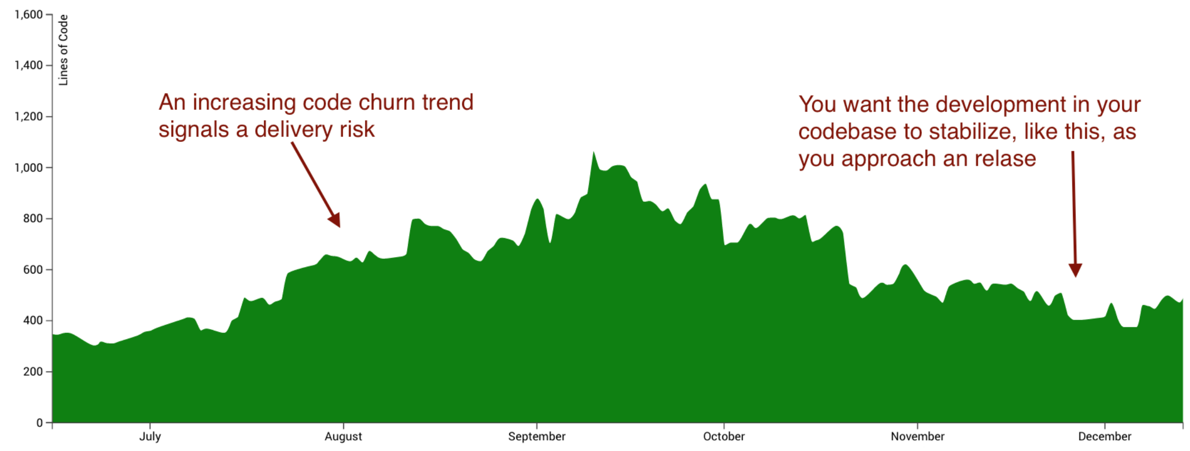


Fig. 2.28: Use code churn to reason about delivery risk.

2.1.6 Code Age

Code Age is a much underused driver of software design. In this guide we'll cover how you interact with the analysis results and how you use the presented information to guide your architectural decisions.

Drive to Stabilize

Code evolve at different rates. As you've learned in the Hotspots Guide (see *Hotspots* (page 12)), some parts of your codebase tend to change much more frequently than others. The *Code Age Analysis* gives you another powerful evolutionary view of your system. It's a view that helps you evolve your codebase in a direction where the system gets easier to maintain and more stable.

The age of code is a factor that should (but rarely do) drive the evolution of a software architecture. In general, you want to stabilize as much code as possible. A failure to stabilize means that you need to maintain a working knowledge of those parts of the code for the life-time of the system.

How do we measure Code Age?

CodeScene measures code age per source code file (or any content, actually). We define the age of code as "the time of the last change to the file". Note that this means *any* change. It doesn't matter if you rename a variable, add a single line comment or re-write the whole module. All those changes are, in the context of Code Age, considered equal.

This definition is fairly rough and in the future we're likely to take the amount of change to a file into account when calculating age. But for now, age is that time since the last change. And the resolution is months.

Inspect your Code Age Distribution

The age distribution graph shows how much of your codebase that you have managed to stabilize.

The example graph in Fig. 2.29 shows a codebase under heavy development. As you see, 20% of the source code files have been modified the past month. Here's how you use this information:

- See how much of the code you manage to stabilize.
- Identify sub-systems that have become commodities.

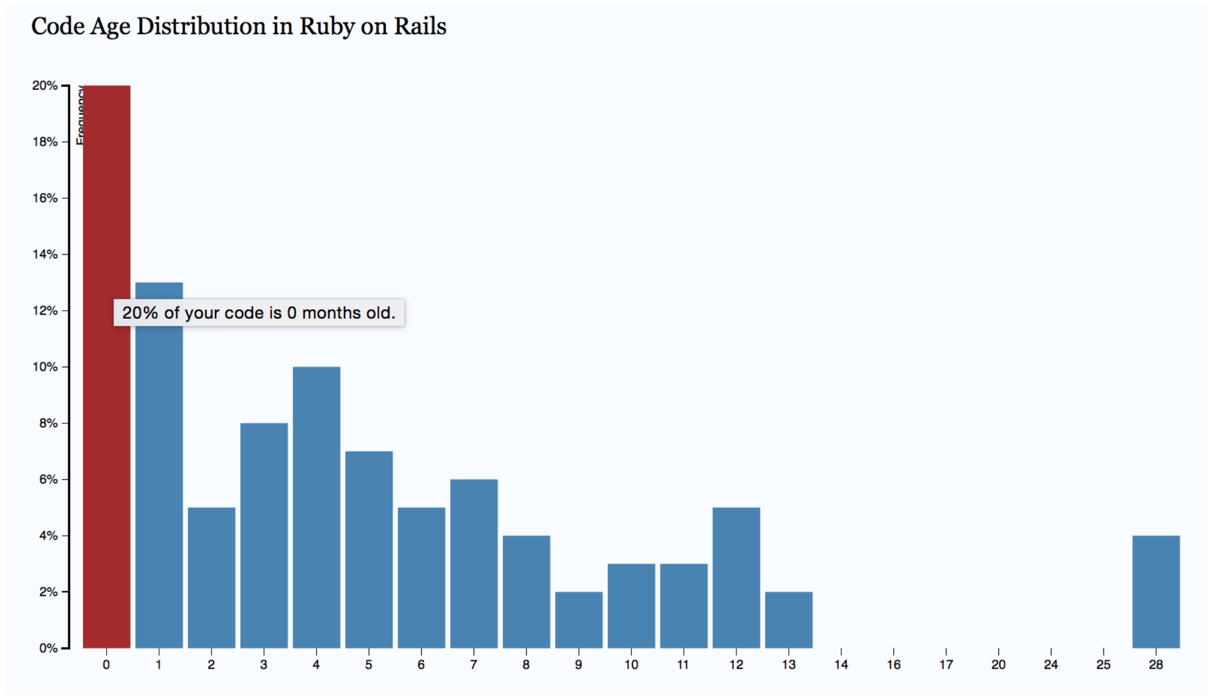


Fig. 2.29: An example of code age distribution.

Let's discuss these two points. First of all, you want to stabilize as much code as possible. Stable code means that its quality is known. It also limits the size of the codebase where a developer has to maintain an active mental model of the code. New code (0-2 months old) is of course where the current development happens and you expect some activity here; a system that doesn't change is a system that no one uses. What you want to look out for is everything in between. That is, the code that's neither particularly old nor do we need to work with it on a monthly basis.

The reason we'd like to avoid having code that is neither old nor new has to do with human forgetting. Such code is old enough that the original programmers are unlikely to remember the details. If we need to dig into code that we no longer remember well, we pay a high price. So please watch out for a codebase where you have a flat distribution.

The second use case for Code Age Distribution is to identify commodities. A commodity is code that's been stable for a long time. You see an example from the development of the Clojure programming language in Fig. 2.30.

This is a good starting point; If you have a lot of code, as in the distribution in Fig. 2.30, that you haven't modified in years, there's an opportunity to drive your software architecture in a leaner direction. To do that we need to get more information. We need to understand *where* in the codebase those stable parts are. That information is provided in the *Age Ring View* that we discuss below. Before we get there, however, we need to be aware of some possible biases.

Possible sources of Bias in the Age Distribution

As noted above, code age is measured since the time of any change to a file. That means, if you reorganize your codebase by moving source code files to different folders, your code will appear much younger than it actually is.

Unfortunately we do not provide a way to counter this bias in the current version of CodeScene. But please stay tuned for future versions where we'll solve this.

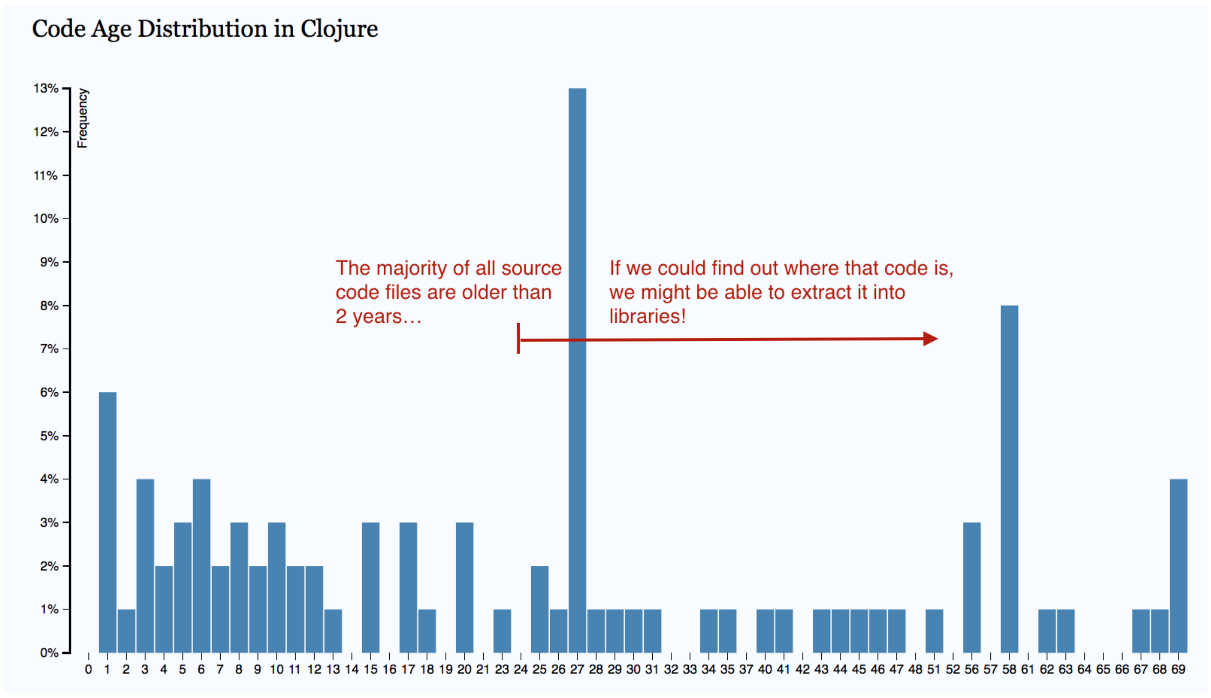


Fig. 2.30: Code age distribution in Clojure.

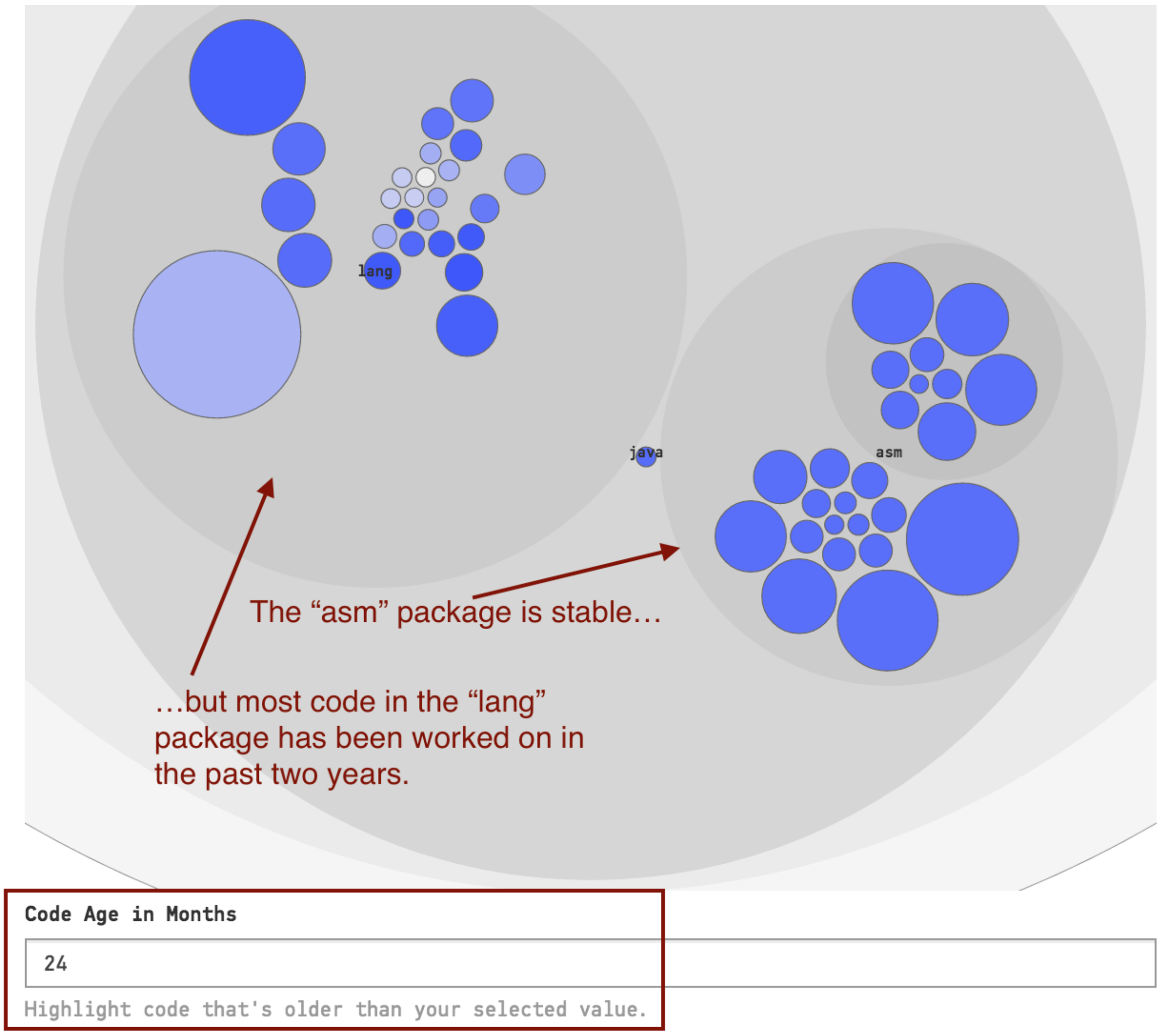
Fig. 2.31: Annual rings of a tree.

Identify Stable and Unstable Sub-Systems in the Age Ring View

So, the Code Age Distribution told us that we've a lot of code that we haven't modified in a long time. The Age Ring View lets you identify where those stable parts are.

Think of the Age Ring View as the annual rings of a tree, but for code. You specify a cut-off point for the age you're interested in and inspect the resulting view.

You select the cut-off point based on the Age Distribution in your codebase. The cut-off point should mean something in your context. For example, we noted above the the Clojure codebase has a lot of code that's older than two years. Fig. 2.32 hows how we find that code.


lang
java
asm
The "asm" package is stable…
…but most code in the "lang" package has been worked on in the past two years.

Code Age in Months

24

Highlight code that's older than your selected value.

Fig. 2.32: Stable code in Clojure.

Extract Stable Packages as Libraries

Once you've found stable packages you may want to consider to extract them as packages. If we transform stable packages into libraries we get a set of advantages:

- *Stable code lets us maintain long-term cognitive models:* The developers now only needs to focus on the API of these packages.
- *Minimize cognitive load for new developers:* As a direct consequence, new developers have less code to understand as they enter your codebase. Age is not something that's visible in the code itself and it's thus hard to know if I, as a developer, have to understand that part of the system or not.

- *Know where extra tests add most value.* You may want to write a set of high-level automated checks around your extracted packages. Those test scripts would capture your understanding of the package and ensure your expectations are correct. Since the code under test is stable, your tests will be stable as well. The reason you have them is so that you can ensure that you don't break existing code when you someday have to modify a part that is a known commodity in your system.
- *Know which tests you don't have to run in each build.* Once you stabilize code, you don't need to run the unit-, function-, integration-tests for that part in every single build. That means you can shorten your delivery cycle by ignoring tests in the parts of the system that haven't been changed for ages.

Identify Parts That Fail to Stabilize

Sometimes you'll find a package (component, sub-system, etc) whose parts change at different rates as in Fig. 2.33

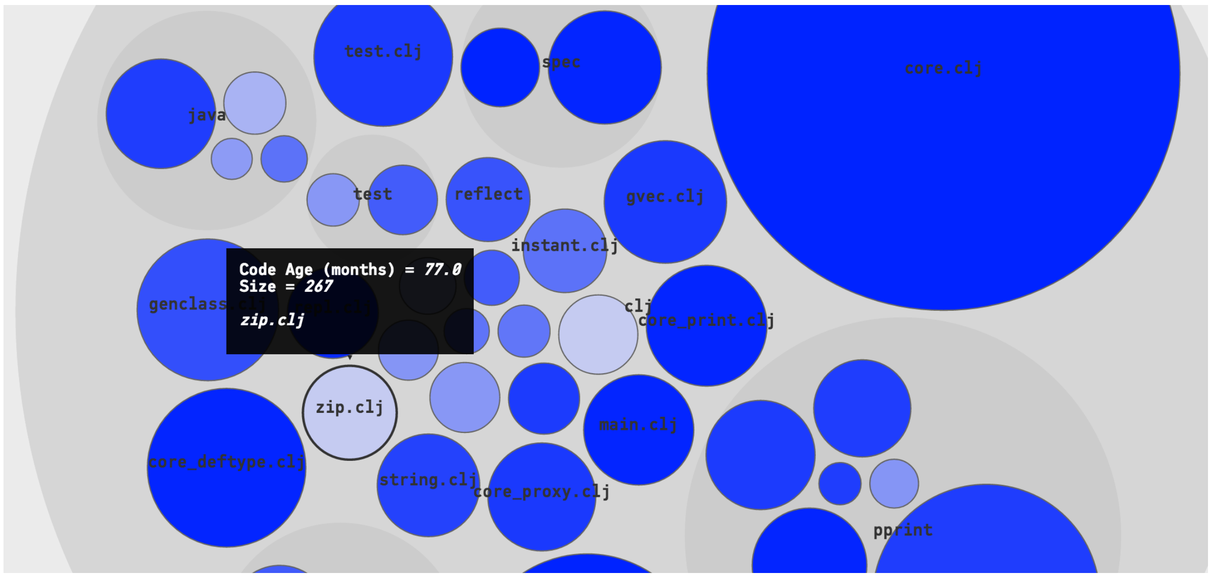


Fig. 2.33: Code that stabilizes at different rates.

Code in the same package/subsystem that change at different rates is a warning sign. It either means that 1) some of the code is of lower quality and we need to patch it often or 2) the parts model different aspects of the problem domain.

Our general recommendation is to try to split packages by the age of the elements contained within them. That is, organize your code by age. Consider the same strategy for larger files that fail to stabilize. Split their content into several, cohesive files. That way, you'll get information on what parts of the problem domain that are volatile and the parts that are stable.

Use Code Age to assess Knowledge Loss

A Code Age analysis has more usages than just software architecture. If you have areas of Knowledge Loss in your codebase you can use Code Age to assess how severe the loss is. Is the abandoned code a part that has been under active development recently? In that case, I would worry. If not, things look better. Sure, you get a knowledge gap with each developer that leaves, but that gap is in a part of the system that you haven't been working on for a long time. Besides, since that code is so old, it's also likely that the original developers, even if they were still present, would have a learning curve themselves.

2.2 Architectural

2.2.1 Architectural Analyses

CodeScene's architectural analyses lets you run Hotspots, Temporal Coupling and more on the *architectural level* of your high level design. The results give you the power to evaluate how well your architecture supports the evolution of your system.

What's an Architectural Component?

An *Architectural Component* is a logical building block in your system. For example, if you build a Microservices architecture, each microservice could be considered a logical block. Similarly, if you organize your code in layers (MVC-, MVP-, MVVM-patterns, etc), each layer would be a logical block.

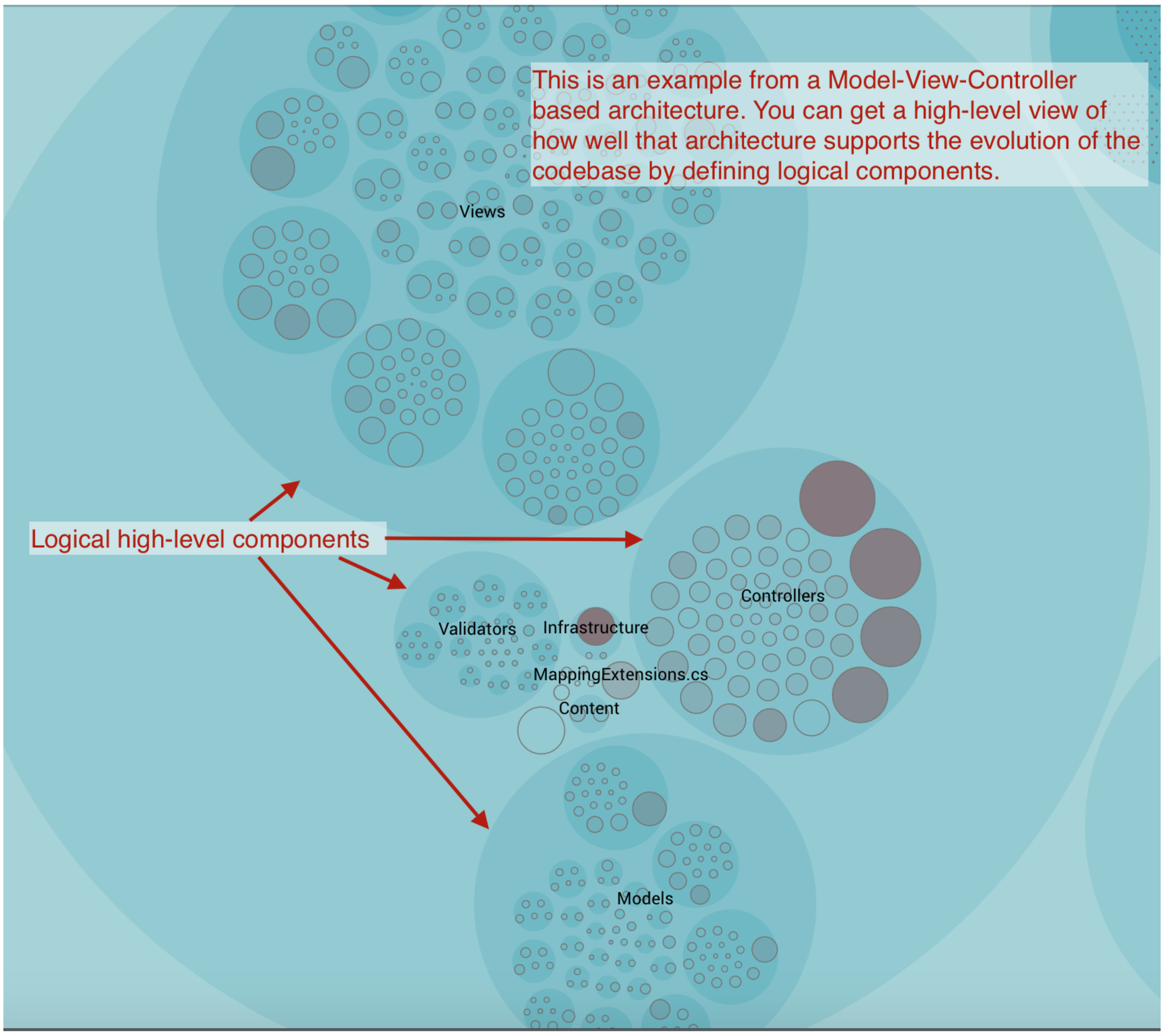

This is an example from a Model-View-Controller based architecture. You can get a high-level view of how well that architecture supports the evolution of the codebase by defining logical components.
Views
Logical high-level components
Controllers
Validators
Infrastructure
MappingExtensions.cs
Content
Models

Fig. 2.34: An example of architectural components.

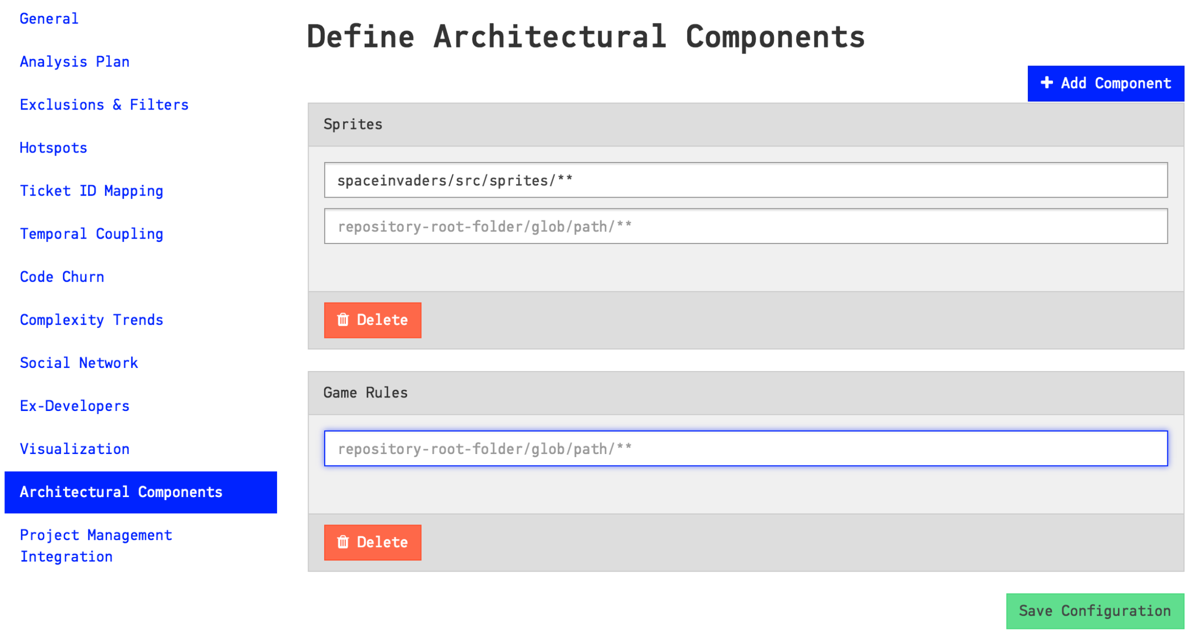
An Architectural Component could also be much more coarse. For example, let's say that you're interested in the co-evolution of your application code versus the test code. Perhaps because you suspect that you spend way too much effort on keeping your automated tests up to date. In that case, you'd define two Architectural Components: *Application Code* and *Automated Tests*.

So, the short answer is that what you consider an Architectural Component depends upon your architectural style. You want to specify components that help you answer the questions you have. For example, do the change patterns in the code match the intent of the architecture? Often, the potential for large maintenance savings are found in these architectural analyses once you spot patterns that violate your architectural principles.

Define your Architectural Components

You need to configure your Architectural Components in order to enable the corresponding analyses.

CodeScene offers flexibility in how you define your components. The tool uses *glob* patterns to identify the files that belong to a specific component as illustrated in Fig. 2.35.



Define Architectural Components

Sprites

```
spaceinvaders/src/sprites/**
repository-root-folder/glob/path/**
```

Game Rules

```
repository-root-folder/glob/path/**
```

Fig. 2.35: Configure architectural components by specifying glob patterns for each logical component.

As you see in the picture above, you need to specify a pattern and the name of your component. All content in your codebase that matches your glob pattern will be assigned to an architectural component with the name you specified. For example, in the configuration above, all content below ‘spaceinvaders/source/sprites’ will be considered as the component *Sprites*. Note that you can provide much more granular filters and, with the power of glob patterns, even match individual files.

Interpret the Architectural Analysis Results

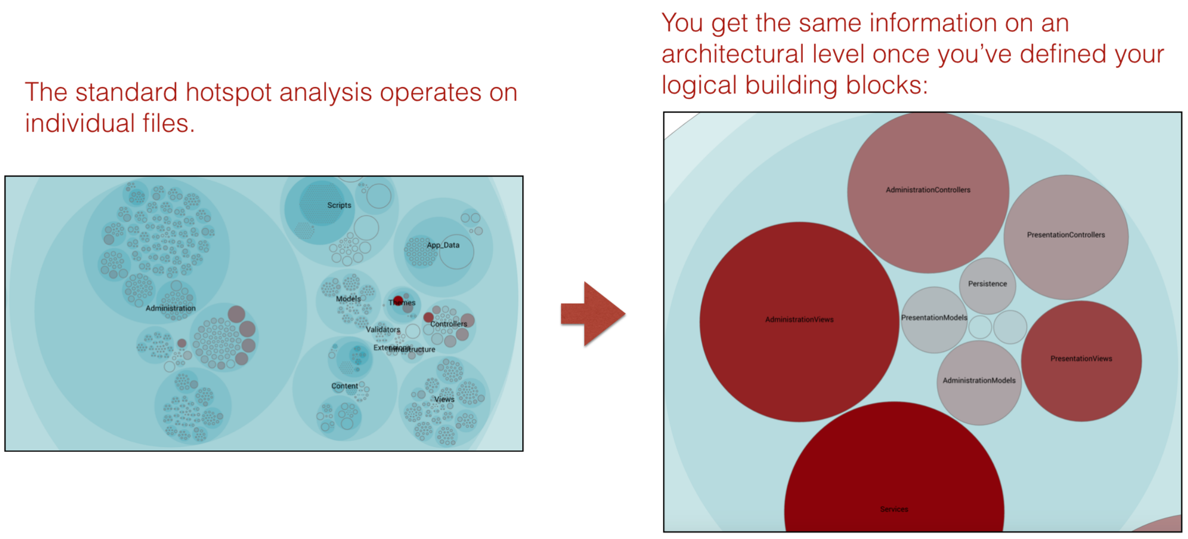
The Architectural Analyses lets you focus on logical building blocks rather than individual files. This allows you to identify architectural Hotspots, as shown in Fig. 2.36.

The architectural analyses also let you identify expensive modification patterns where code changes ripple through multiple logical components, as seen in in Fig. 2.37.

2.3 Social

2.3.1 Social Networks

The *Social Network Analysis* gives you a heuristic on the coordination needs between developers on different teams. The idea is based on Conway’s law - a project works best when its organizational structure is mirrored in software. Using the *Social Network Analysis*, you now have a way to ensure that your organization matches the way the system is designed with respect to the work the developers do.


The standard hotspot analysis operates on individual files.

You get the same information on an architectural level once you've defined your logical building blocks:

Fig. 2.36: Using the hotspot analysis for architectural components.

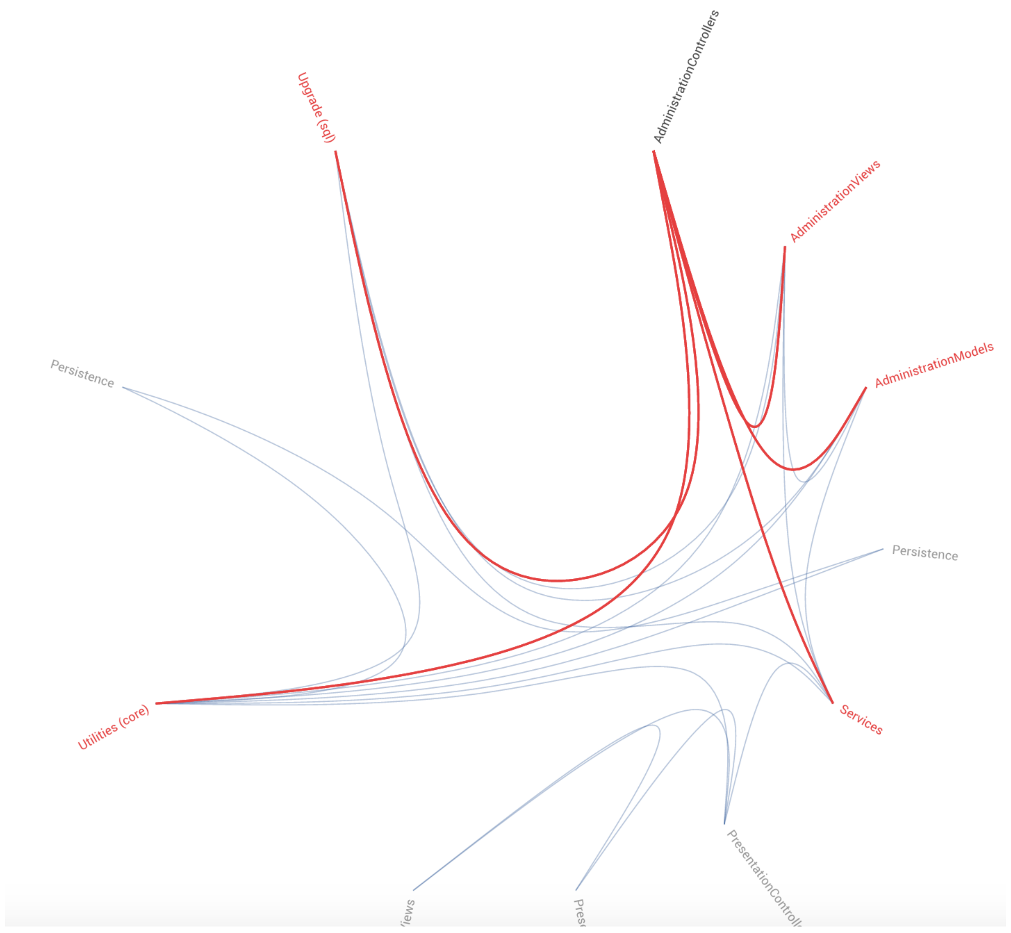


Fig. 2.37: Temporal coupling between architectural components.

The Social Network Is Build from How the Code Evolves

The social network paths are mined from how your codebase is developed. You see an example of a social network in code in Fig. 2.38.

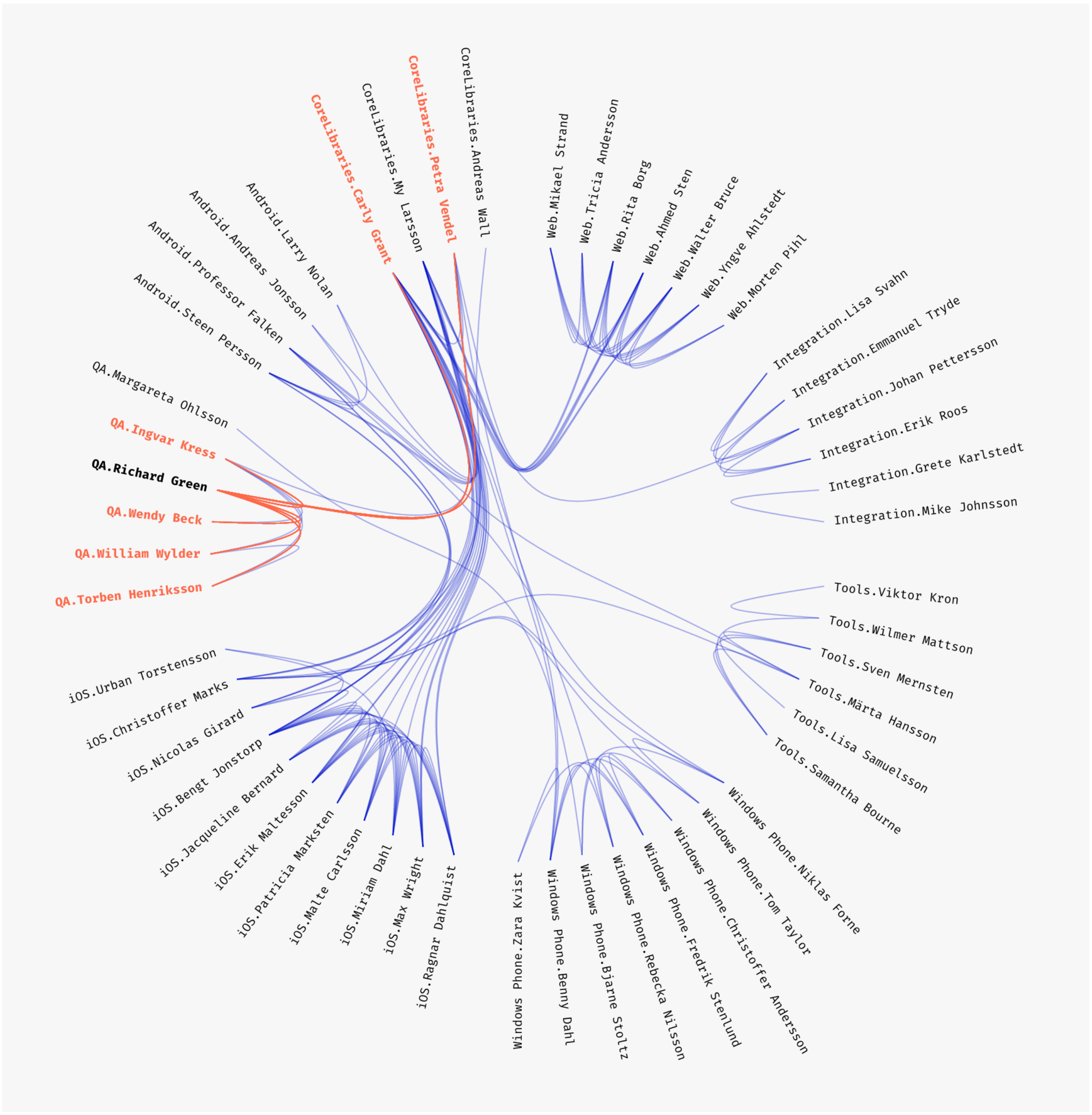


Fig. 2.38: An example of a social network in code.

The network is built by identifying developers that repeatedly work in the same parts of the code. The more often they work in the same parts of the code, the stronger their link in the network. Note that CodeScene filters developers with weak links since they would clutter the visualization (you can change the threshold as described in *Project Configuration* (page 58)).

Define Your Development Teams

The social network lets you identify developers that should be close from an organizational perspective. The visualization in Fig. 2.38 shows an example of an organization with 8 development teams. If you hover over a developer, you highlight their peers that tend to work in the same parts of the codebase. You use this information to evaluate how well your organization supports the way the codebase evolves.

That also means you want to compare your organizational chart with the information in the generated social code network. Any discrepancies has to be understood.

Align Your Architecture and Organization

In a perfect world most of your communication paths would be between developers on the same team. That is, the teams have a meaning from an architectural perspective; People on the same team work on the same parts of the codebase. They share the same context, know each other and have a much easier time coordinating their work.

However, sometimes the world looks radically different. Have a look at Fig. 2.39.

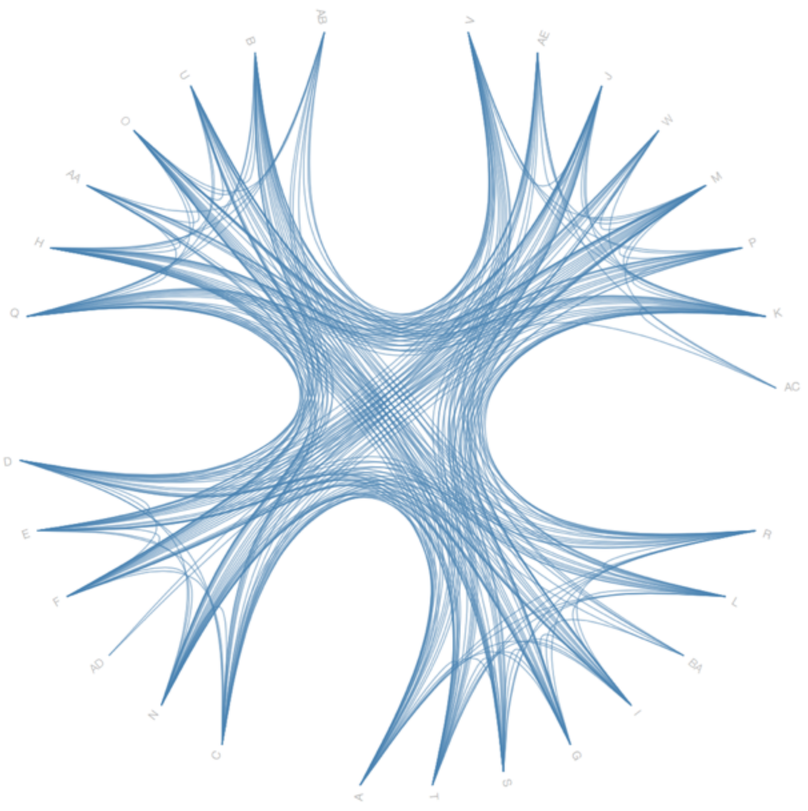


Fig. 2.39: An example of a social network anti-pattern.

The visualization in Fig. 2.39 shows an organization with severe coordination problems. Since the data has been made anonymous to protect the guilty, you cannot read the names of the teams or developers. But you still see that the organization has four teams with a high degree of inter-team coordination between virtually every developer. In practice, this isn't an organization with four different teams. Rather, it's an organization with one giant team of 29 developers with artificial organizational boundaries between them. The resulting process loss due to coordination needs is likely to be severe and lead to inefficient development, quality issues and code that's hard to evolve.

2.3.2 Knowledge Distribution

Let's face it - software development is a social activity. We work in teams, sometimes distributed, where we need to communicate and coordinate in order to solve our tasks. Building an organization responsible for creating and evolving a system is a necessity as soon as your codebase has grown beyond a certain size. It's our way to scale and be able to take on larger problems than what we could as individuals.

But moving from individual developers to teams does not come free; No matter how efficient we, as an organization, are, we'll always pay a price. The cost of team work is known as *process loss*. Process loss is the theory that a team, just like a mechanical machine, cannot operate at 100 percent efficiency. In the mechanical world we have inefficiencies like friction and heat loss. Our software equivalents are coordination and communication. The main challenge in most software projects is to minimize the process loss. Failures to do so often come off as technical issues, when in reality those issues have social roots.

The software industry has been aware of these issues. But until now, we've never had a way to measure them. This is about to change. In this guide you'll learn how CodeScene helps you uncover knowledge distribution and identify team productivity bottlenecks in your system. With the following suite of analyses you're now able to make organizational decisions based on data from how you've actually worked so far.

How Do We Measure Knowledge?

The knowledge metrics are based on the amount of code each developer has contributed. CodeScene looks at the deep history of each file to calculate contributions. This makes sense for two different reasons:

1. The last snapshot of a source code file wouldn't be good enough since such shallow ownership is sensible to superficial changes (e.g. re-formatting issues, automated renaming of variables, etc).
2. Even if one developer completely rewrites a piece of code, its original author will still retain some knowledge in that area since they're familiar with the problem domain. The metrics in CodeScene acknowledge that and will retain some knowledge for the original developer as well.

CodeScene uses the name of each committer to calculate knowledge metrics. So please *make sure* you understand the possible biases discussed in the guide *Know the possible Biases in the Data* (page 51).

Prepare the Analyses by Assigning Colors to Developers and Teams

Your knowledge maps are based on colors to give you an accessible high-level overview. Please refer to the guide in *Configure Developers and Teams* (page 65) to prepare for the knowledge analyses.

Tip: Use the *Code Churn by Author* analysis results to identify the top contributors so that you can assign them as distinct colors as possible.

Explore the Individual Knowledge Map

The first knowledge analysis measures the knowledge distribution for individual developers in your codebase.

Each developer is assigned a color in the following visualization. The color of each file represents its main developer (that is, the developer who has contributed most of the code). You see the resulting visualization in Fig. 2.40.

All knowledge maps are interactive:

- Click on a circle to zoom in on the corresponding package.
- Click outside the circle to zoom out.
- Hover the mouse over a circle to see information about the package or file it represents.

Explore your Team Knowledge Maps

CodeScene also measures knowledge distribution on a team level and this information is usually even more valuable than the individual metrics.

As soon as you've assigned developers to a team, as described in *Configure Developers and Teams* (page 65), CodeScene will accumulate their individual knowledge into their teams. The analysis results

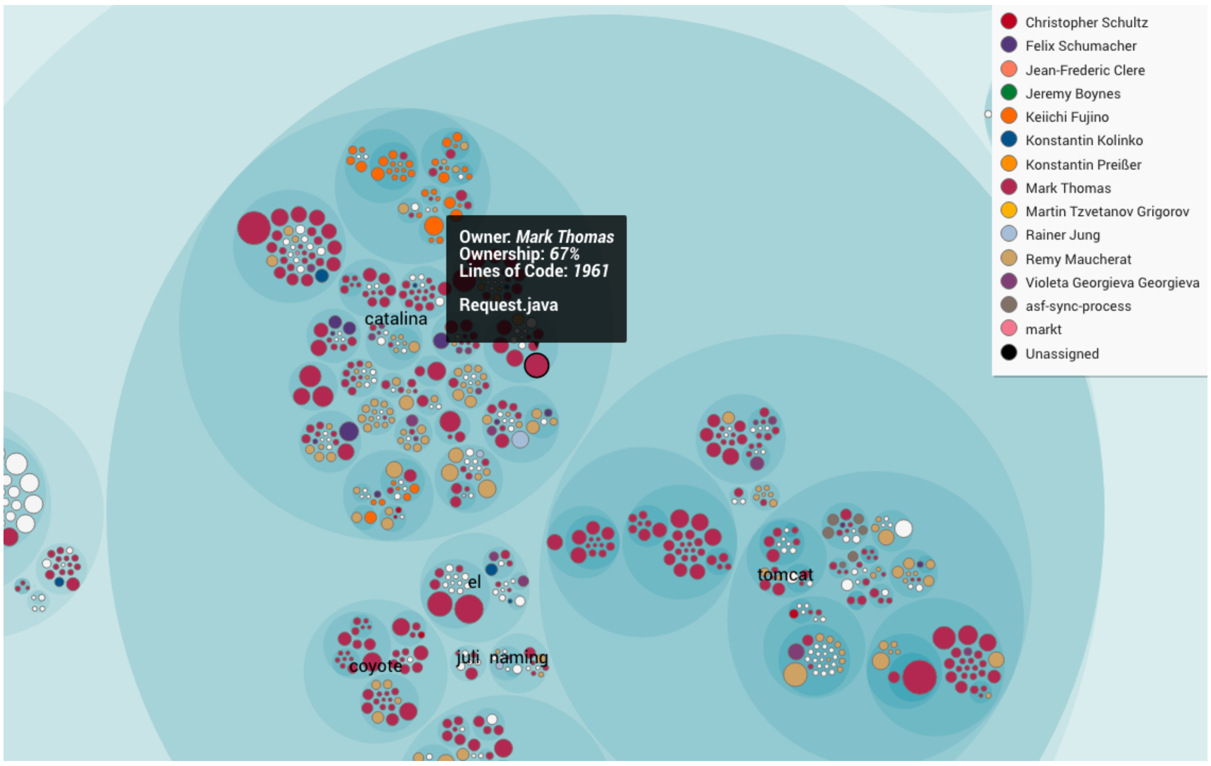


Fig. 2.40: An example of a knowledge map, hovering a circle to get more information.

are presented using the same principles as for the Individual Knowledge Map. Only now, each color represents a team.

The Team Knowledge Map lets you reason about both the responsibilities of the different teams. In general, you want to ensure that your team organization is reflected in the software architecture of your system. Please refer to the discussions in the guide *Social Networks* (page 41) for more information on the organizational theories and how they correlate to the quality and efficiency of your organization.

Uncover the Knowledge Loss in your Codebase

Knowledge loss represents code that is written by a developer who is no longer part of your organization or project. You use this information to reason about the knowledge distribution in your codebase and as part of your risk management since it is an increased risk to modify code we no longer understand. In addition, you can also use the analysis pro-actively to simulate the consequences, in terms of knowledge loss, of planned organizational changes.

The *Knowledge Loss* analysis will accumulate the contributions of all developers that you have marked as Ex-Developers in your configuration (see *Configure Developers and Teams* (page 65)). Those parts of the codebase that are dominated by Ex-Developers are marked as red in the knowledge loss visualization. Fig. 2.41 shows an example from an organization where some core developers have left.

2.3.3 Parallel Development and Code Fragmentation

Large scale software development is a social activity. However, the technical nature of our work tends to obscure that fact and we often mistake organizational issues for technical problems.

One such example is excess parallel development. Excess parallel development is something that happens when your architecture cannot support the way you're organized. You may have 20-30 developers that need to modify the same file, but for different reasons. The symptoms you see are often technical, for example expensive merges, code that's hard to understand since it's changed by different people all the

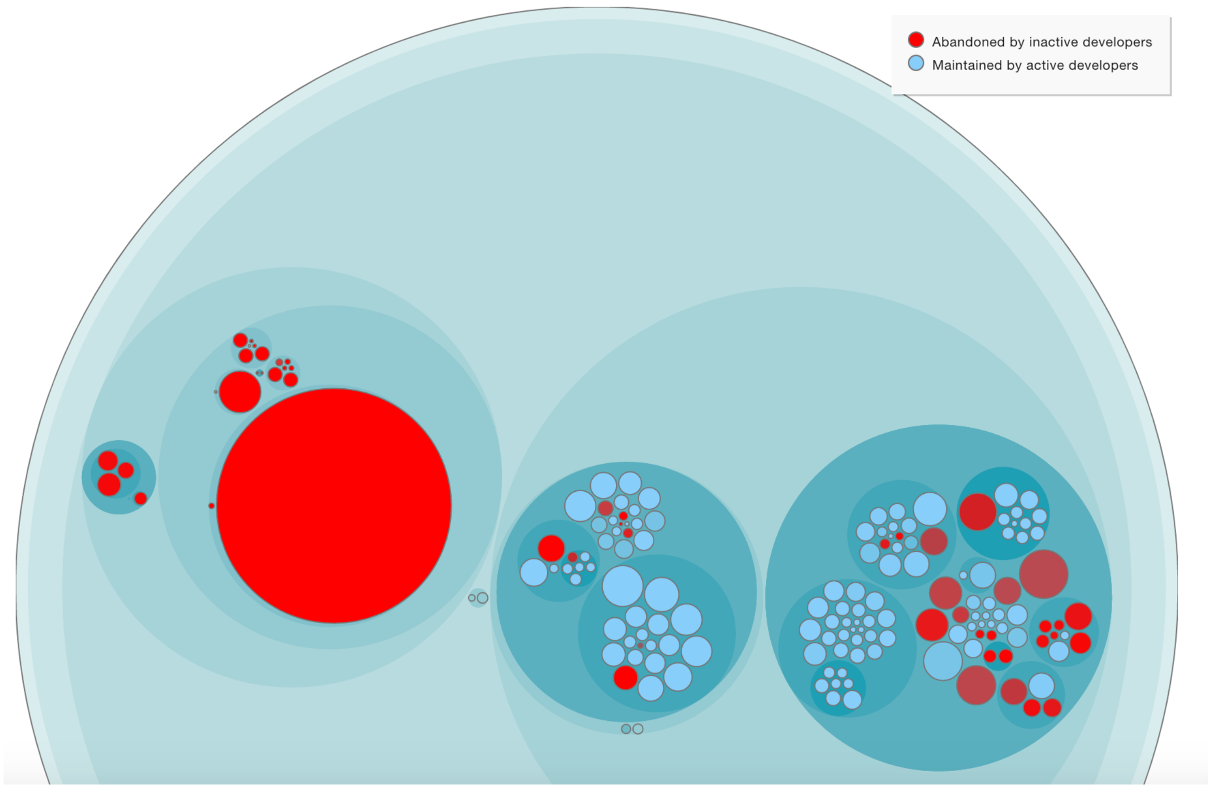


Fig. 2.41: An example on a knowledge loss analysis.

time, or unexpected feature interactions. CodeScene's *Parallel Development* analysis helps you uncover and prioritize these problems.

Before you read on, please note that CodeScene uses the name of each committer to calculate the fragmentation metrics. So *make sure* you understand the possible biases discussed in the guide *Know the possible Biases in the Data* (page 51).

The Fragmentation View Uncovers Excess Parallel Development

Excess parallel development means the modules have a high *fragmentation value*. A high fragmentation value means that the development effort is shared between multiple programmers. This is a risk you want to be aware off - the number of programmers is one of the best predictors of the number of post-release defects in a module. The more programmers, the more quality issues in that code.

CodeScene runs the fragmentation analyses on both individual authors and teams. You may want to focus on the team view in case you have cohesive teams with well-defined responsibilities. If that's not the case, start with investigating the fragmentation by authors. Both of them shows the fragmentation value of each file in the codebase, as illustrated in Fig. 2.42.

The fragmentation map in Fig. 2.42 shows the *fractal value* of each file. A fractal value is the degree of parallel work:

1. Fragmentation 0 (zero): This means that the file has had a single developer working on it.
2. Fragmentation closer to 1.0 (one): The closer to 1.0 the fragmentation gets, the more developers behind the code and the smaller the contribution of each developer.

Once you've found a part of your codebase with excess parallel work you want to get more detailed information. The Fractal Figures described in the next section gives you all the details you need.

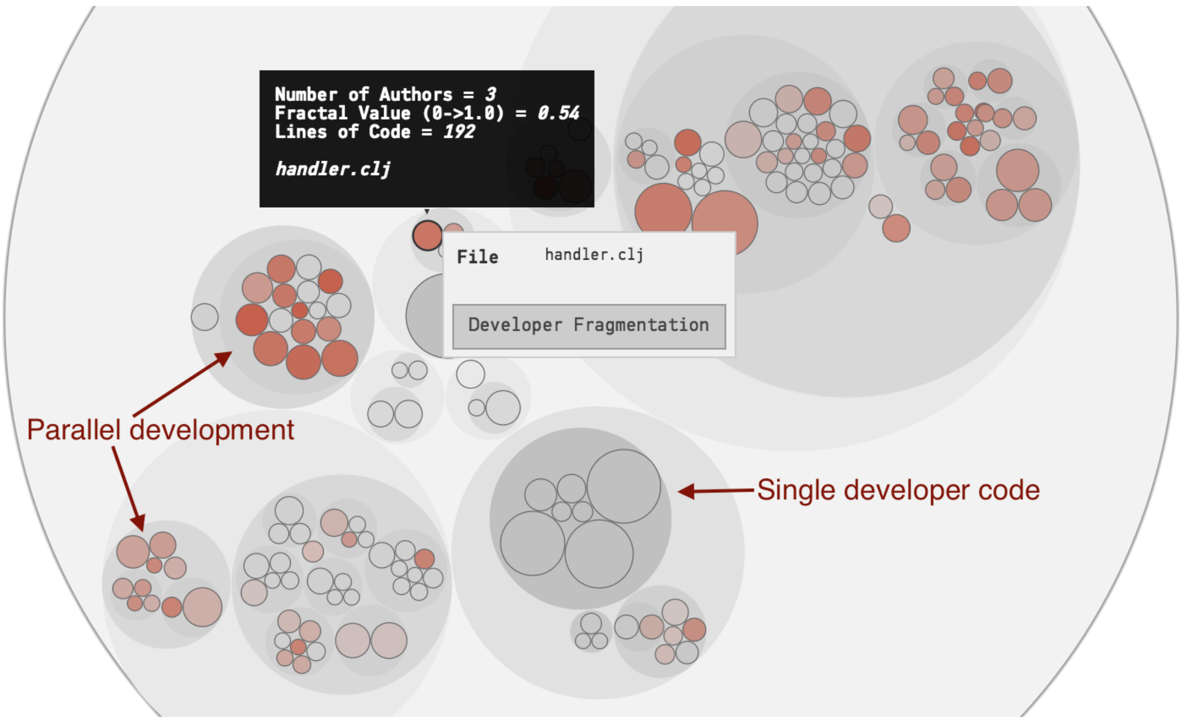


Fig. 2.42: The fragmentation map shows files with excess parallel development.

Get more Detailed Information with Fractal Figures

The fragmentation map in Fig. 2.42 is interactive. That means you can click on each file and inspect the amount of fragmentation as illustrated in Fig. 2.43.

2.3.4 Modus Operandi

Modus Operandi is the method of opereration. It's the signature for how you work with the codebase and lets you discover trends and risks in the type of coding you do.

What is Modus Operandi?

Forensic psychologists refer to Modus Operandi as the method of operation, a criminal signature. Software teams have a modus operandi, too. Our analyses help you uncover it to better understand how the team works. The information will never be precise, but lets you ask the right questions and guide your discussions by opening a new perspective on your daily work.

Inspect Trends in Your Commit Messages

CodeScene's JIRA integration (see *Project Management Analyses* (page 51)) lets you discover trends in the type of work you do. However, not all organizations use JIRA. There may also be work-related information that isn't available in JIRA.

Thus, CodeScene provides a second data source for work-related trends: your commit messages. Your commit messages is an interesting data source too, as illustrated in Fig. 2.44

By default, CodeScene will identify all commit messages that contain the texts *bug*, *fix*, or *defect* as illustrated in Fig. 2.45. Please note that the matches are always case insensitive. That is, if you specify *bug*, CodeScene will match both *bug* and *Bug*.

You can configure CodeScene to match any word or phrase that you want. You just specify a regular expression in the Modus Operandi section of the project configuration.

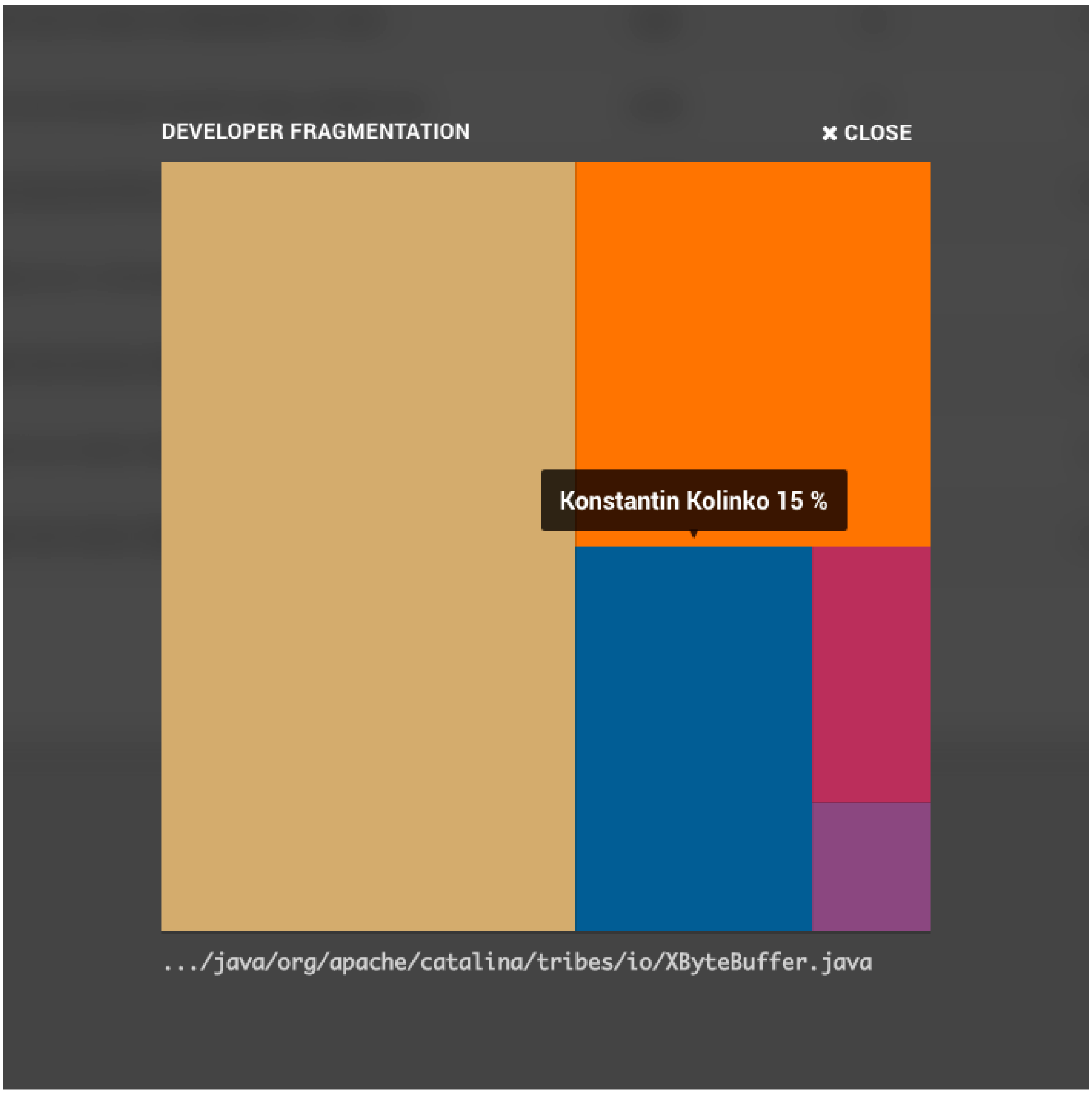
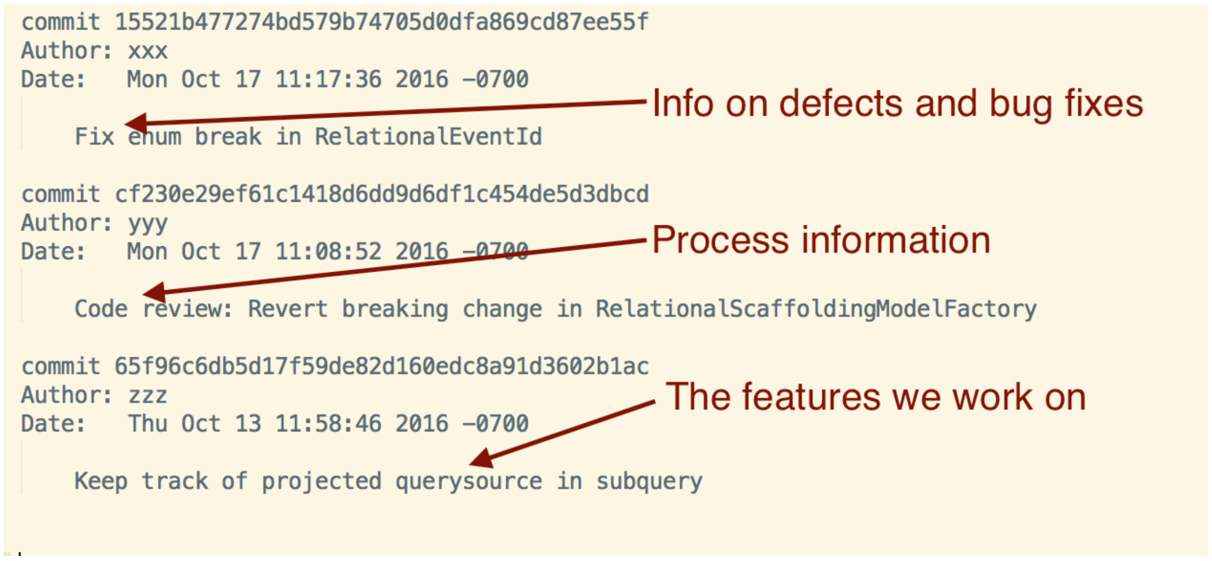
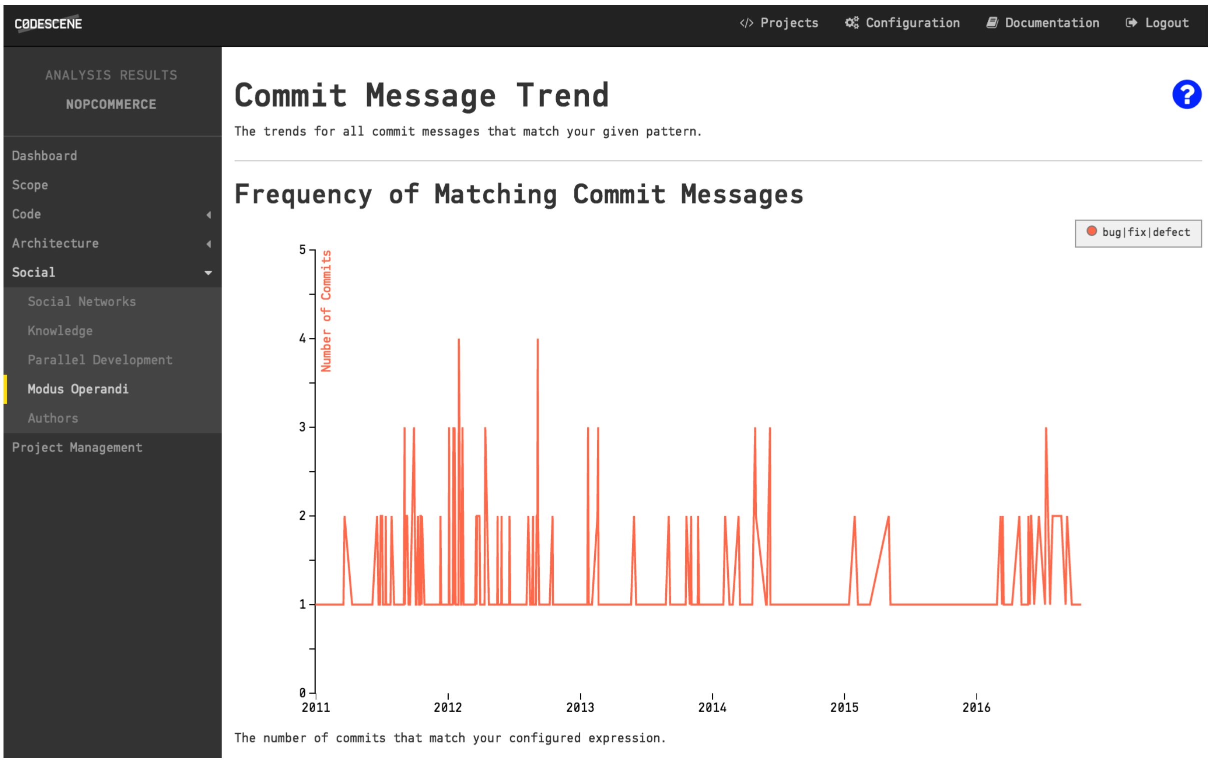


Fig. 2.43: An example of a developer fragmentation. Hovering a colored fragment shows the developer and the relative contribution.



```
commit 15521b477274bd579b74705d0dfa869cd87ee55f
Author: xxx
Date:   Mon Oct 17 11:17:36 2016 -0700

    Fix enum break in RelationalEventId

commit cf230e29ef61c1418d6dd9d6df1c454de5d3dbcd
Author: yyy
Date:   Mon Oct 17 11:08:52 2016 -0700

    Code review: Revert breaking change in RelationalScaffoldingModelFactory

commit 65f96c6db5d17f59de82d160edc8a91d3602b1ac
Author: zzz
Date:   Thu Oct 13 11:58:46 2016 -0700

    Keep track of projected querysource in subquery
```

Fig. 2.44: Your commit messages contains work-related information.



Commit Message Trend

The trends for all commit messages that match your given pattern.

Frequency of Matching Commit Messages

The number of commits that match your configured expression.

Fig. 2.45: Inspect all commits that mention a particular word or phrase.

2.3.5 Know the possible Biases in the Data

Our social metrics, like all software metrics, are an approximation of the real world. There will always going to be corner cases and biases in the data. In particular, there are some situations where the metrics don't perform as well. So please read the following section in order to minimize the bias in the analysis results.

Developers with Multiple Aliases

A developer may end up with multiple aliases. Perhaps they're committing from both a personal- and a company account. Or they've changed their e-mail address. This introduces a bias in the data since CodeScene uses the name of each developer as their identification.

Fortunately, this bias is easy to avoid by utilizing a Git feature called `.mailmap`. A `.mailmap` is a file that you include in the root of your Git repository. The file specifies a mapping from multiple names and addresses to the canonical name and address of each developer with multiple aliases. It's straightforward to use a .mailmap, so please check out the git log documentation for the format.

Autosquash Commits

Some teams may use a Git feature called *autosquash*. This feature is a way of re-writing the development history. It may be fine if squashing is used for the work of an individual developer. Unfortunately the feature is sometimes used to combine the work of multiple programmers into a single commit.

The consequence is that the analyses lose important data for temporal coupling and, in particular, the social metrics become more limited than they'd have to be. For example, it's not possible to generate a knowledge map over individual programmers, which means that you miss the opportunity to use the analysis methods for on- and off-boarding.

It's highly recommended that you reconsider the autosquash strategy in case you apply it today. In general, the work of multiple programmers should not be compressed in a single commit.

Pair Programming

The knowledge metrics in CodeScene are based on the author of the code as recorded by Git. This may obviously be misleading if your organization does pair-programming.

A future version of CodeScene will address pair-programming knowledge by having the option to fetch the author names from the commit message instead and assign the knowledge to both authors. However, in the current version there's no such option.

If you use pair-programming you're also likely to rotate pairs on a daily basis. In that case, we recommend that you ignore the analysis of individual developers and focus on your team knowledge map instead, which will have accurate data.

2.4 Project Management

2.4.1 Project Management Analyses

CodeScene's suite of project management metrics let you measure where you spend your costs and inspect both cost and activity trends.

The Need for Project Management Metrics

CodeScene's project management metrics answer two common questions:

1. How shall we prioritize improvements to our codebase?

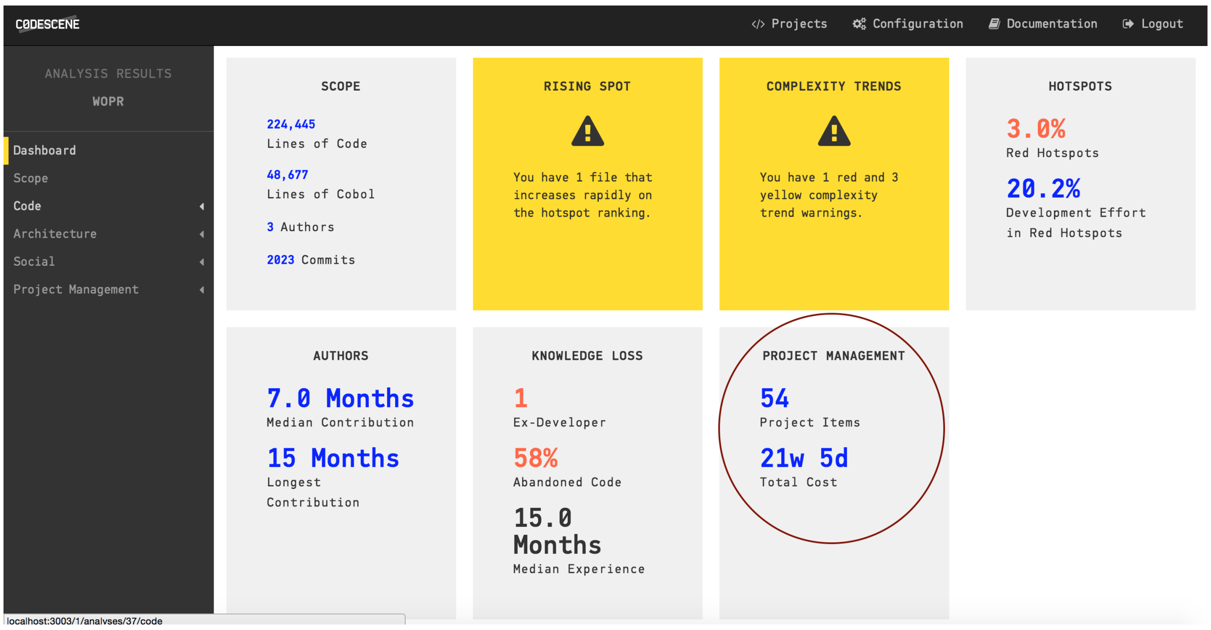
2. How can we follow-up on the effects of the improvements we do?

Sure, our Hotspot analysis already addresses these questions and gives us a tool to prioritize. However, there's a linguistic chasm between developers and managers here; To a manager, a "commit" doesn't carry much meaning. A commit is a technical term that doesn't translate to anything in the manager's world. At the same time, technical debt and low quality code are important subjects to address. So how can we talk the language of a manager while still tying our data back to something that communicates with the developers responsible for the code?

CodeScene bridges this chasm by introducing a suite of project management metrics. These metrics combines our existing version-control measures with data from Jira. This gives you Hotspots measured by cost rather than the more technical change frequency metric. It also gives you trends in both your costs and the type of work you do (e.g. features vs maintenance). Let's see how it is done.

Learn to Interpret the Project Management Metrics

You need to configure CodeScene to access the Jira service. Once that's done, CodeScene will automatically retrieve the Jira data and run an analysis on it. The results are presented on the Analysis Dashboard as shown in Fig. 2.46.



SCOPE
224,445 Lines of Code
48,677 Lines of Cobol
3 Authors
2023 Commits

RISING SPOT
You have 1 file that increases rapidly on the hotspot ranking.

COMPLEXITY TRENDS
You have 1 red and 3 yellow complexity trend warnings.

HOTSPOTS
3.0% Red Hotspots
20.2% Development Effort in Red Hotspots

AUTHORS
7.0 Months Median Contribution
15 Months Longest Contribution

KNOWLEDGE LOSS
1 Ex-Developer
58% Abandoned Code
15.0 Months Median Experience

PROJECT MANAGEMENT
54 Project Items
21w 5d Total Cost

Fig. 2.46: The metrics are accessed from the analysis dashboard.

Click on the Project Management tile to get to the detailed results. The detailed results presents Hotspots by cost and let you access the trends. Let's look at some examples.

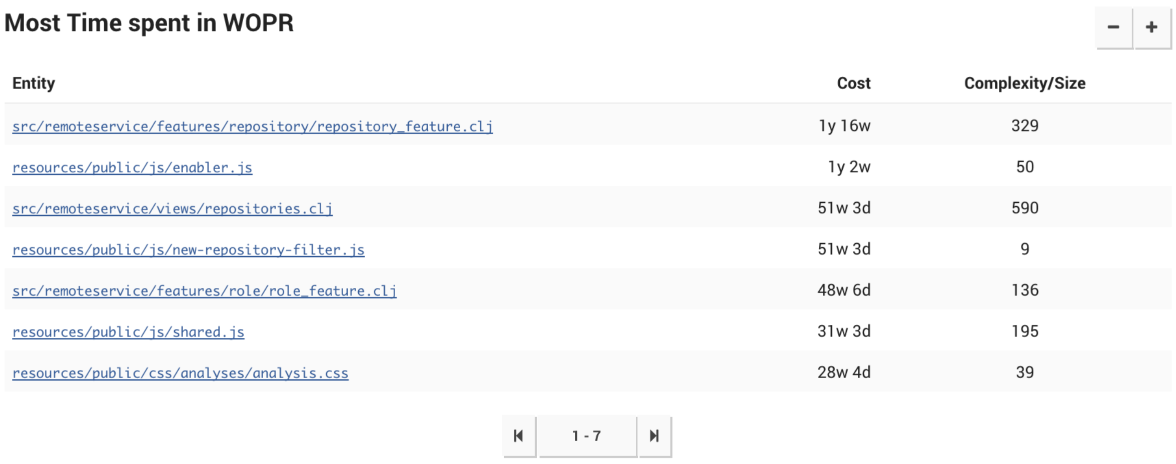
The Hotspots by Costs provide an overview of which part of the code that are the most expensive to maintain. The analysis works just like the normal technical Hotspot analysis. The main difference is that these Hotspots are ranked by cost rather than the change frequency of the code. You see an example in Fig. 2.47

As you see in Fig. 2.47, most time is spent in a module named *repository_feature.clj*. That means you want to prioritize improvements to that code. Before you do that, however, you'd like to look at the cost trend to see if this is recently accumulated cost or if the Hotspot has been expensive to maintain for a long time.

You access the cost trends by clicking on a Hotspot and select *Cost Trend* as shown in Fig. 2.48.

The cost trend is presented in two different graphs:

1. The first graph will show the accumulated cost by month for the selected Hotspot. The costs are a summary of all Jira issues that have involved work in this specific Hotspot.

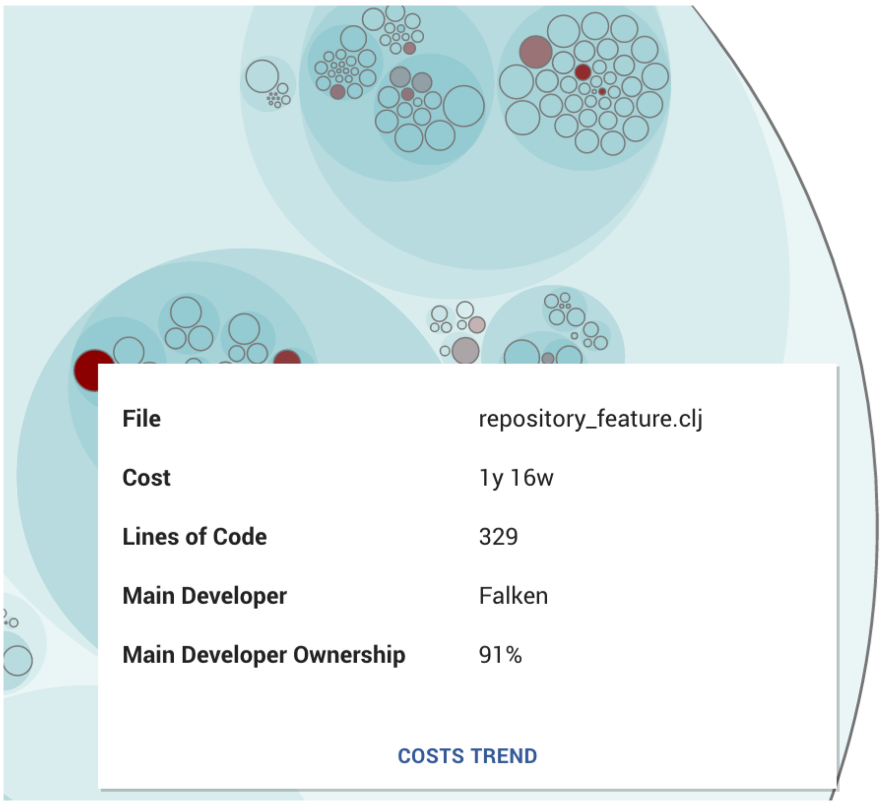


Most Time spent in WOPR

Entity	Cost	Complexity/Size
src/remoteservice/features/repository/repository_feature.clj	1y 16w	329
resources/public/js/enabler.js	1y 2w	50
src/remoteservice/views/repositories.clj	51w 3d	590
resources/public/js/new-repository-filter.js	51w 3d	9
src/remoteservice/features/role/role_feature.clj	48w 6d	136
resources/public/js/shared.js	31w 3d	195
resources/public/css/analyses/analysis.css	28w 4d	39

1 - 7

Fig. 2.47: A Hotspot analysis by cost lets you see where you spend most time.



File	repository_feature.clj
Cost	1y 16w
Lines of Code	329
Main Developer	Falken
Main Developer Ownership	91%

Fig. 2.48: Access the cost trends from the interactive Hotspots Map.

2. The second graph shows the cost distributed across the type of work you've done.

You use this information to ensure that the code evolves in the right direction. For example, you'd like to see a decrease in the amount of bugfixes and an increase in the amount of feature related issues. You can also use the cost trends to measure the effect of large-scale improvements as illustrated in Fig. 2.49.

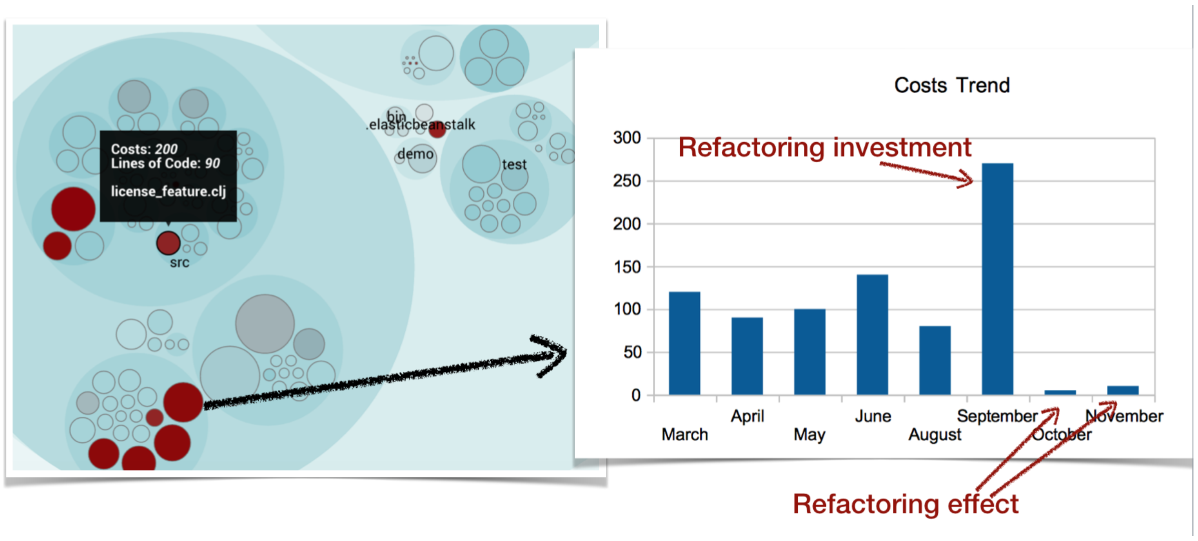


Fig. 2.49: Use the Cost Trends to measure the effect of improvements.

A Note to Developers

You'll probably notice a high correlation between the project management results and the results from the technical Hotspot analysis. This is an expected finding. However, the project management metrics have another usage. Since the project management metrics speak the language of a non-technical managers, these analyses provide a basis for communication. Use this data to motivate investments in software quality, like for example to explain the need for a larger refactoring of one ore more top Hotspots.

Pre-Requisites for the Project Management Analyses

This suite of analyses fetches data from a project management tool like Jira. CodeScene provides a Jira integration as a separate service. However, the Jira data only contains the raw costs (hours, story points, etc) of a story - there's no specification of how those costs are shared across the different parts of your codebase.

CodeScene solves this problem by mapping the Jira data to our wealth of version-control metrics. There are a number of pre-requisites that are mandatory for this process to work:

- You need to include your Jira Ticket/Issue/Story ID in the commit messages. We use that information to unify the data sources.
- You need to have a cost metric in your Jira story. CodeScene supports time-based costs (i.e. minutes of time to completion) and story points.

Limitations in the Analysis Data

The cost trends and analysis results will never be better than the available raw Jira data. That is, if your reported costs on a Jira story are too far off, the analysis don't have any way to adjust it.

In addition, there are a number of limitations that you need to be aware of:

- The total costs for a Jira issue are assigned to the last known month that the issue was worked on. So if you have long-running issues, you'll see the costs assigned to a single month even if the issue took, let's say, 3 months to implement.
- All files that were worked upon in a Jira issue get assigned the same cost. In reality, some files typically account for a larger amount of the total costs, but there's no way for CodeScene to know that. Instead we treat each file as an equal contributor to the issue.

In general, you'll find that you get much more out of the analysis results as long as you remember that the project management metrics are heuristic in their nature rather than precise predictions of the future.

2.4.2 Risk Analysis

CodeScene analyses the risk of each commit. This lets us present both a risk trend and also an early warning as soon as a high risk commit is detected.

You use this information to react early and focusing code reviews and testing. You also use the overall risk trend as input and feedback on planned delivery activities.

How Does CodeScene Know That a Commit is High Risk?

CodeScene calculates a unique *risk profile* for your codebase. The risk profile is based on how the system has evolved and what a typical change looks like. That is, CodeScene looks more at how a commit looks than the changed code itself.

CodeScene's risk profile is a combination of technical and social metrics. The technical metrics relate to the amount of code that is changed, how many different files that are changed, and the diffusion of the changes (e.g. how many different sub-systems does the commit touch).

The social dimension of the risk profile relates to the experience of the programmer doing the change. The more experienced the programmer, the lower the risk. This means that two commits with identical changes may be classified differently depending on the programmer who made the change; Experience mediates risk. For example, if I make a large sweeping change to the Linux kernel, my change probably has higher risk than an identical change made by Linus Torvalds. Please note that *experience* is relative to your codebase and measured as how much each programmer has contributed to your code historically.

The risk classificaiton that you'll see in CodeScene always combines these technical and social dimensions.

What's the Scale of Commit Risks?

CodeScene scores each commit on the range 1 to 10. 1 is a low risk change and 10 is the highest risk. By default, CodeScene flags all commits with a risk of 7 (or higher) as high risk. You can change this threshold in the project configuration.

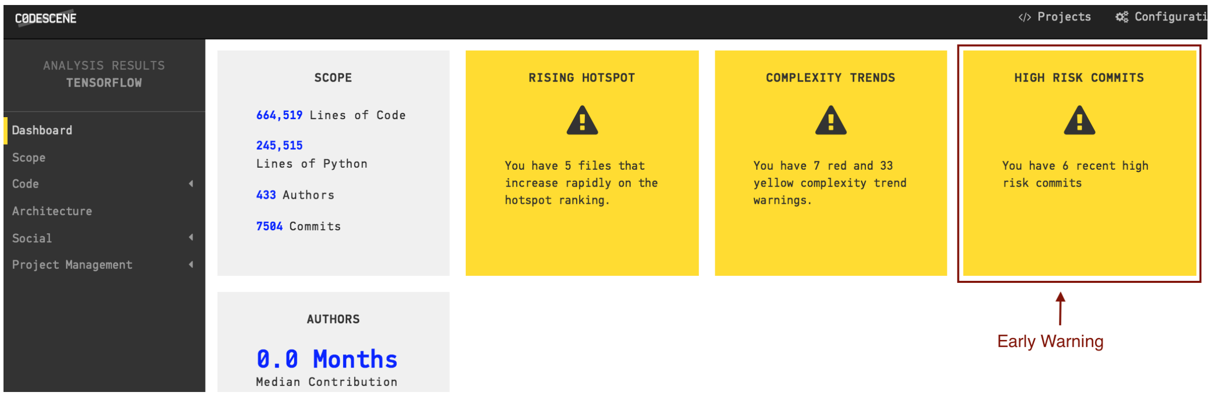
Inspect your Risk Profile

CodeScene delivers an early warning as soon as a high risk commit is detected as illustrated in Fig. 2.50.

Click on the early warning shown in Fig. 2.50 to view the commit details as illustrated in Fig. 2.51.

CodeScene also calculates a rolling average of your risk profile. This analysis lets you reason about risk trends in your project and relate that trend to both your ongoing work as well as predict delivery risk.

The example in Fig. 2.52 shows a project where there's a significant increase in the average risk during development. When you see a trend like this it's important to understand *why*. Perhaps several large features are being implemented? Or perhaps there's a change in the ways of working or development methodology? In any case, it would probably be a mistake to plan a release in July for this particular project since there has been a lot of recent high risk work that deviates from how the codebase grew before that date.



SCOPE

664,519 Lines of Code

245,515 Lines of Python

433 Authors

7504 Commits

RISING HOTSPOT

You have 5 files that increase rapidly on the hotspot ranking.

COMPLEXITY TRENDS

You have 7 red and 33 yellow complexity trend warnings.

HIGH RISK COMMITS

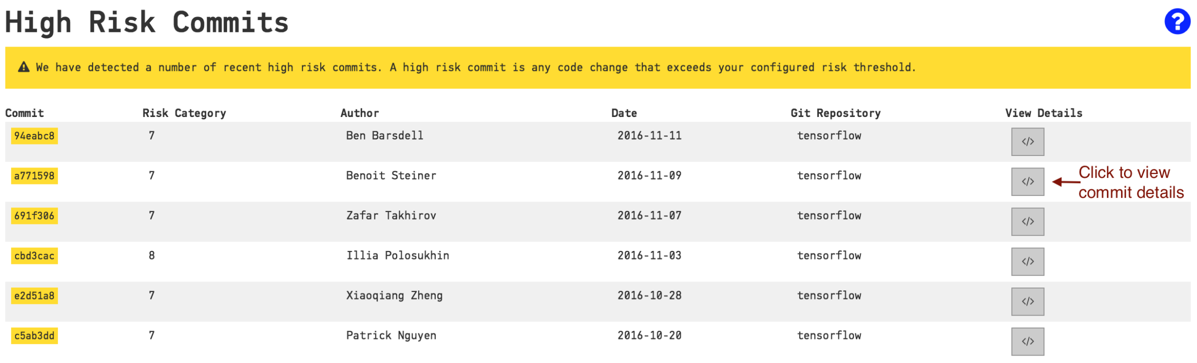
You have 6 recent high risk commits

AUTHORS

0.0 Months

Median Contribution

Fig. 2.50: An Early Warning for recent high risk commits.



High Risk Commits

We have detected a number of recent high risk commits. A high risk commit is any code change that exceeds your configured risk threshold.

Commit	Risk Category	Author	Date	Git Repository	View Details
94eabc8	7	Ben Barsdell	2016-11-11	tensorflow	</>
a771598	7	Benoit Steiner	2016-11-09	tensorflow	</>
691f306	7	Zafar Takhirov	2016-11-07	tensorflow	</>
cbd3cac	8	Illia Polosukhin	2016-11-03	tensorflow	</>
e2d51a8	7	Xiaoqiang Zheng	2016-10-28	tensorflow	</>
c5ab3dd	7	Patrick Nguyen	2016-10-20	tensorflow	</>

Fig. 2.51: Inspect the details of each recent high risk commit.

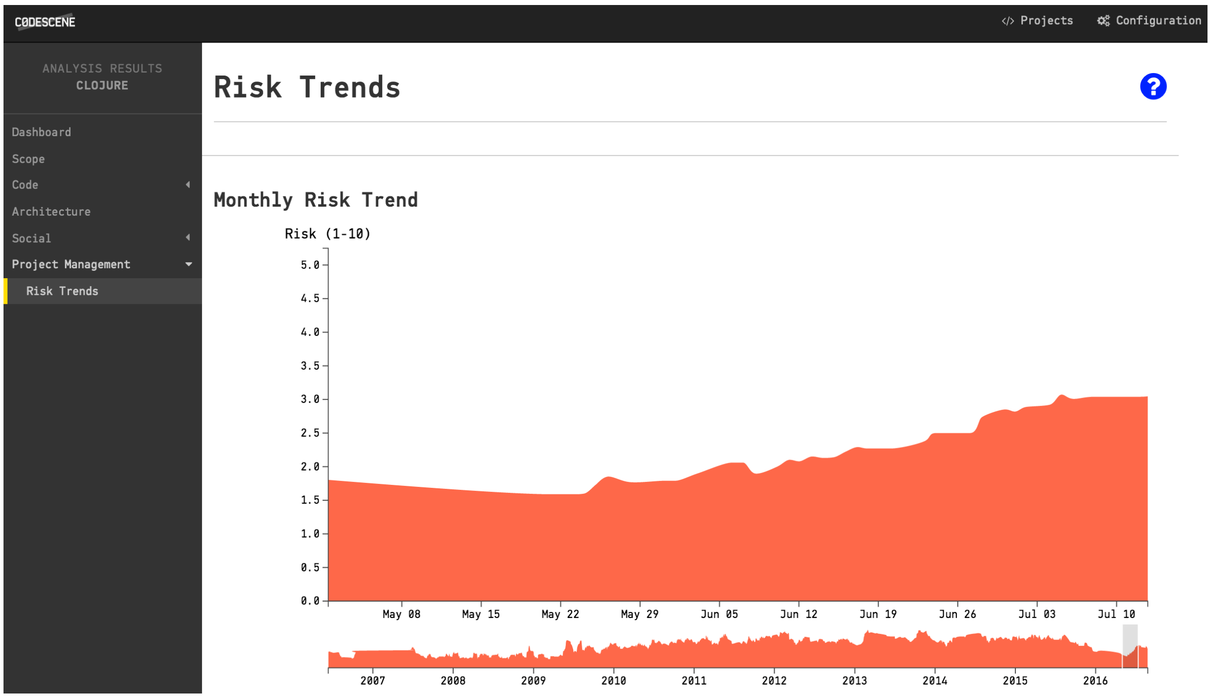


Fig. 2.52: The risk trend shows the average risk in the evolution of your codebase.

Risks Are Relative To The Analysis Period

It's important to note that your risk profile is always relative to your particular analysis period. That is, you get a different risk profile if you analyze the complete history of your code versus a short retrospective analysis. This is by design and most likely to be the information you want.

However, you need to be aware that if you run a Retrospective analysis, you may see *more* high risk commits. That just means those commits stand-out compared to the rest of the work you did in that sprint/iteration; It doesn't necessarily mean that those commits would be high risk relative to the complete evolution of your system. To find out, you need to run a full analysis.

Chapter 3

Configuration

3.1 Project Configuration

3.1.1 Specify the Git Repository to Analyze

Your first step is to tell CodeScene where your code is. You have three different ways of doing that:

1. Specify the paths to your local, physical Git repository, which has to be on the same machine as CodeScene runs on. The path you specify has to be to the root folder of your repository (i.e. the folder that contains your .git folder).

2. Let CodeScene scan a folder on your file system for repositories to analyze. You'll be prompted with the results and are free to ignore the repositories you want to exclude. This option is useful in a multi-repository project.

3. Specify the URLs to Git remotes. CodeScene supports the protocols specified by Git clone: ssh, http, and git. CodeScene will clone the remotes to a local folder that you specify in the configuration as illustrated in Fig. 3.1. Note that CodeScene will re-use a local Git repository if there's an existing clone on the path you specify. Also note that you need to have a an ssh-key that lets the CodeScene (system) user access your remote repositories.

Finally, note that you cannot mix local repository paths with URLs to remote Git repositories in a single analysis project.

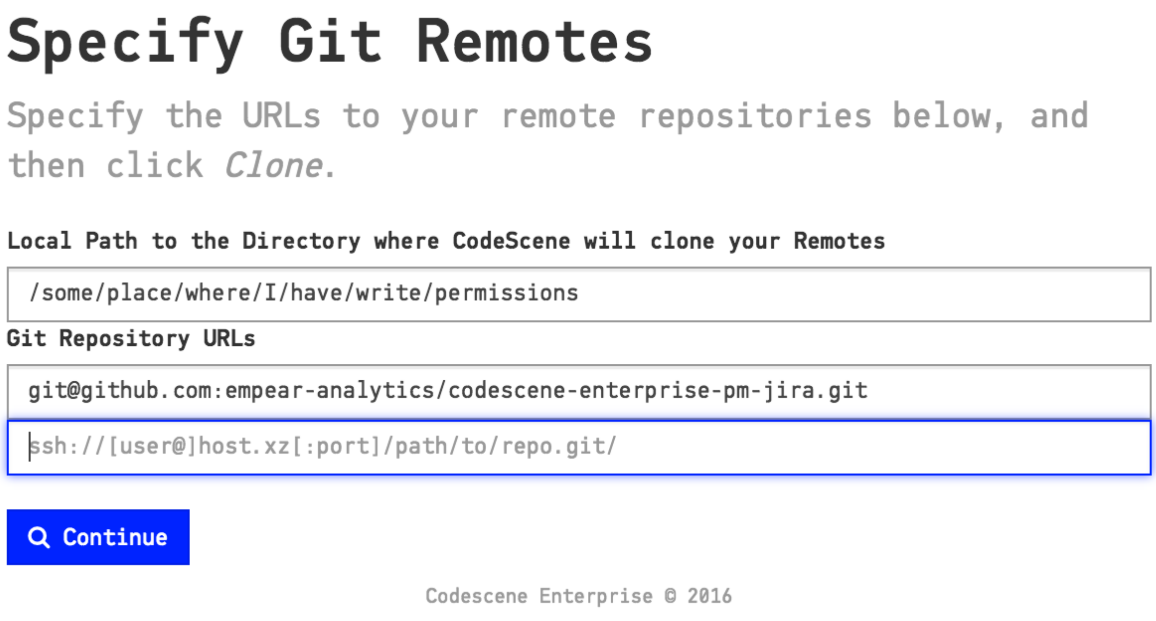
3.1.2 Analyze Projects organized in Multiple Git Repositories

There's a recent trend towards organizing the source code of larger systems in multiple Git repositories. For example, you may have the code for your user interface in one repository, the code for your service layer in another repository and perhaps even a Git repository dedicated to your back end mechanism. Another typical example is *Microservices* where each service is deployed according to its own life cycle. In that case, organizations often chose to use one Git repository per service.

CodeScene supports an analysis of multiple repositories at once. All you have to do is to specify the paths to them:

The screenshot above shows three repositories that belong to the same product. During an analysis, CodeScene will analyze the evolution of the code in all those repositories *as though they were in the same physical Git repository.*

You can specify as many repositories as you want and remove one at any time (just erase the text in that box). However, a word of warning: do *NOT* attempt to analyze unrelated repositories in the same configuration. First of all it's a breach of the license agreement. Worse, you won't get useful results since many of the basic metrics, like Hotspots, are relative metrics.



Specify Git Remotes

Specify the URLs to your remote repositories below, and then click *Clone*.

Local Path to the Directory where CodeScene will clone your Remotes

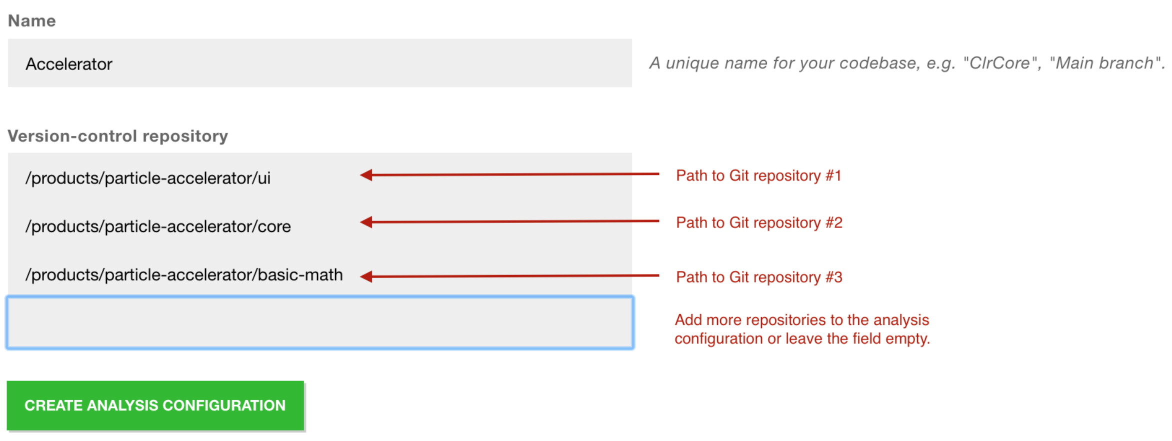
/some/place/where/I/have/write/permissions

Git Repository URLs

git@github.com:empear-analytics/codescene-enterprise-pm-jira.git

ssh://[user@]host.xz[:port]/path/to/repo.git/

Fig. 3.1: Let CodeScene clone your Git repositories through their URL.



Name

Accelerator

A unique name for your codebase, e.g. "ClrCore", "Main branch".

Version-control repository

/products/particle-accelerator/ui

/products/particle-accelerator/core

/products/particle-accelerator/basic-math

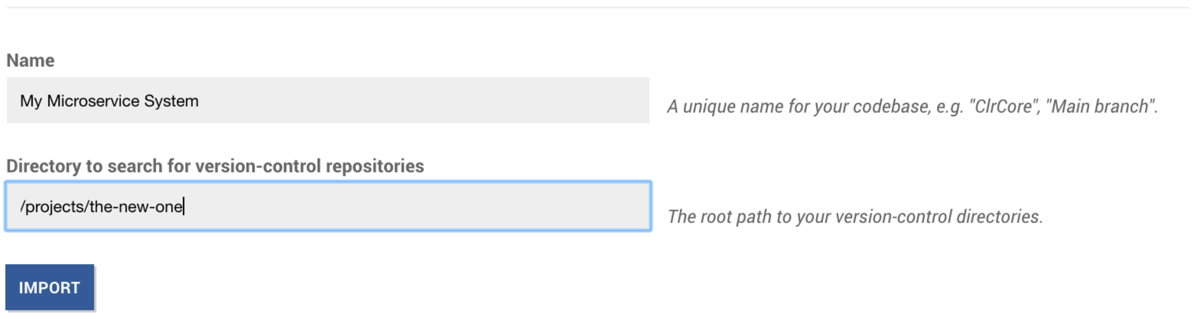
Path to Git repository #1
Path to Git repository #2
Path to Git repository #3
Add more repositories to the analysis configuration or leave the field empty.
CREATE ANALYSIS CONFIGURATION

Fig. 3.2: Configuration of multiple repositories.

3.1.3 Auto-Import Repository Paths

Specifying one or two repositories by hand is straightforward. However, some systems consists of hundreds of repositories. In that case you want to use the auto-import feature.

The auto-import feature lets you specify a root path to where your repositories are located. Here's what it looks like:



Name

My Microservice System

A unique name for your codebase, e.g. "ClrCore", "Main branch".

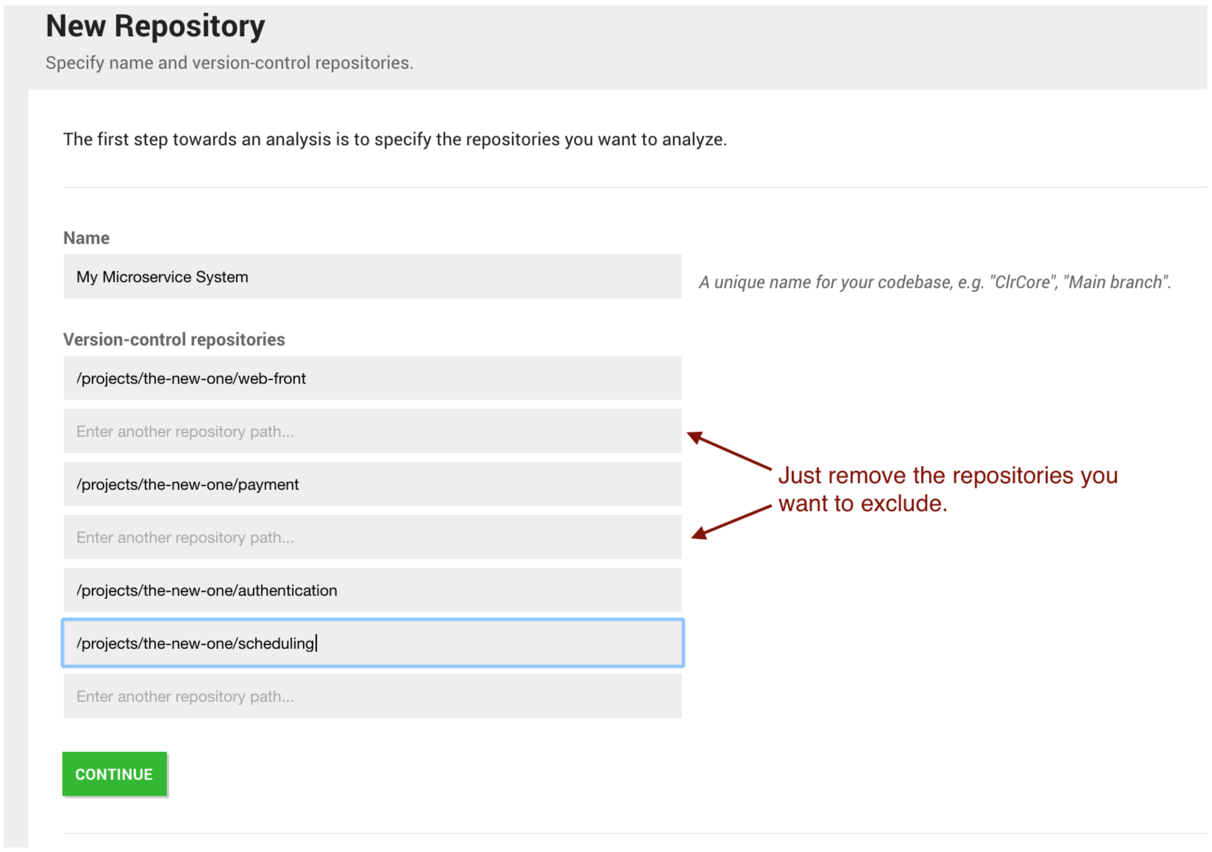
Directory to search for version-control repositories

/projects/the-new-one

The root path to your version-control directories.

Fig. 3.3: Automate the import of multiple repositories.

CodeScene will scan the path you provide to discover any Git repositories. The discovered Git repositories are presented in a list. Note that you can add additional repositories manually or remove the once you want to exclude:



New Repository

Specify name and version-control repositories.

The first step towards an analysis is to specify the repositories you want to analyze.

Name

My Microservice System

A unique name for your codebase, e.g. "ClrCore", "Main branch".

Version-control repositories

/projects/the-new-one/web-front

Enter another repository path...

/projects/the-new-one/payment

Enter another repository path...

/projects/the-new-one/authentication

/projects/the-new-one/scheduling

Enter another repository path...

Fig. 3.4: The result of auto importing multiple repositories.

From here you just press Continue to proceed with the configuration of your analysis. The rest of the workflow is identical to the case where you specify repositories manually.

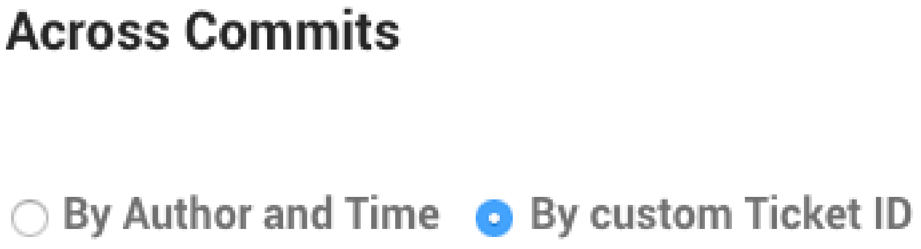
3.1.4 Measure Temporal Coupling across Multiple Repositories

The normal temporal coupling metric considers two files coupled if they tend to change in the same commits. This won't work if your codebase is split across multiple repositories. Instead, you want to aggregate individual commits into logical commits. CodeScene supports two different strategies for aggregating commits:

By Author and Time When you specify this option, the tool will consider all commits by the same author on the same day as a single, logical commit. This option is a heuristic that works well in the absence of a Ticket ID in your data.

By custom Ticket ID This option uses an identifier in your commit headers. All commits that refer to the same identifier will be considered one logical commit.

The second option, *By custom Ticket ID*, is the preferred method. Fig. 3.5 shows the options in the repository configuration section *Temporal Coupling.*

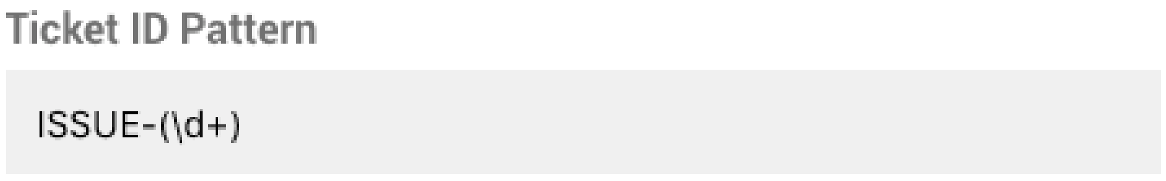


Across Commits

By Author and Time [✓ By custom Ticket ID]

Fig. 3.5: There are two available strategies for aggregating commits.

To aggregate by custom Ticket ID, you need specify a *Ticket ID Pattern*, in the *Ticket ID Mapping* section (see Fig. 3.6). The pattern is used to extract the Ticket ID from the commit message. The example pattern in Fig. 3.6 will extract all identifiers that start with the text `ISSUE-` followed by at least one digit. For example, the commit message `ISSUE-42` will result in `42` as the extracted Ticket ID.



Ticket ID Pattern

ISSUE-(\d+)

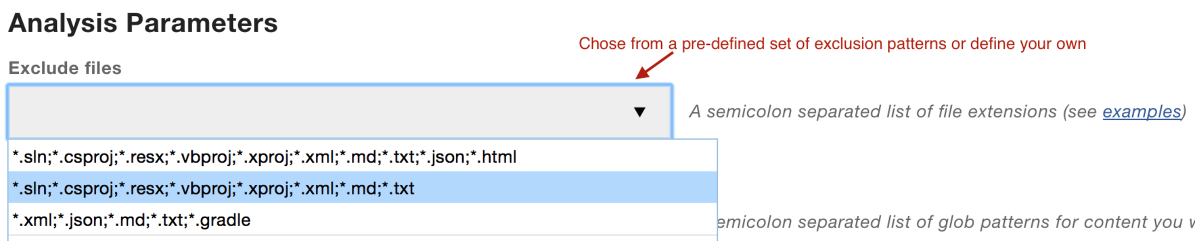
Fig. 3.6: Configure a pattern to extract a Ticket ID.

Note that CodeScene will still calculate normal temporal coupling on a single commit basis. You want that in order to spot unexpected dependencies between files in the same repository. The temporal coupling results for the logical commits discussed above are presented in a separate analysis view.

3.1.5 Exclude Files from an Analysis

An analysis will include all textual content in your repository. That means: you get an analysis of your build scripts, resource files, configuration files, test data, etc. While it's a good practice to run an analysis of all content every now and then, there's also the risk that you get too much noise in the analysis results. For example, you typically want to exclude auto generated content.

The *Exclude Files* option lets you specify a set of file extensions that will be excluded from your analysis:



Analysis Parameters

Exclude files

Chose from a pre-defined set of exclusion patterns or define your own

A semicolon separated list of file extensions (see examples)

```
*.sln;*.csproj;*.resx;*.vbproj;*.xproj;*.xml;*.md;*.txt;*.json;*.html
*.sln;*.csproj;*.resx;*.vbproj;*.xproj;*.xml;*.md;*.txt
*.xml;*.json;*.md;*.txt;*.gradle
```

emicolon separated list of glob patterns for content you v

Fig. 3.7: Exclude specific types of files.

CodeScene comes with a set of pre-defined exclusion patterns that should match the most common cases. You're free to extend this set if you have additional file types that you want to exclude. Just remember to use a semi-colon (;) to separate each file extension you want to exclude.

3.1.6 Exclude Specific Files and Folders from an Analysis

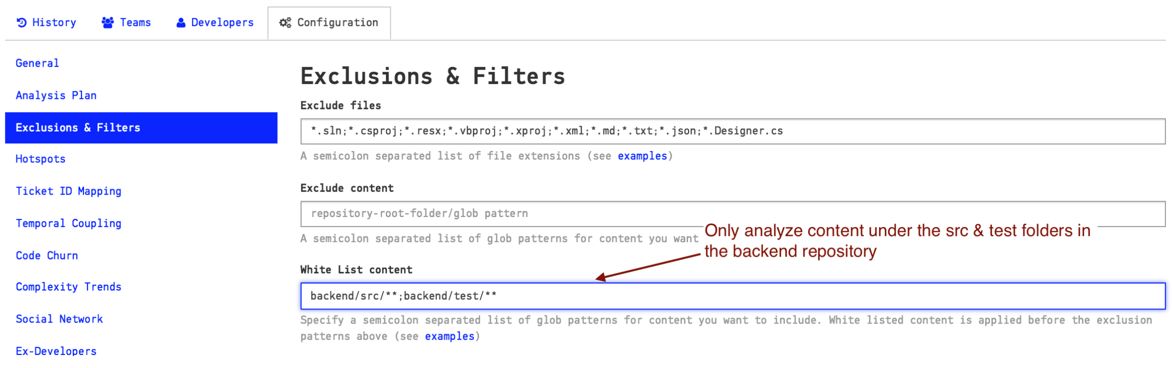
You just learned how you can exclude certain types of files, no matter where they are located in the your codebase. But sometimes you'd like to exclude a particular file or, more often, a complete folder. For example, let's say that you check-in third party code in your repository. You don't want that code to obscure potential analysis findings in your own code.

There are two different ways to exclude complete folders and files:

1. White list the content you want to include in the analysis. All other content will automatically be excluded.
2. Black list the content you want to exclude.

You can specify both white- and black list content. The white listing will be applied first.

You specify a *glob pattern* to white list the content to include in your analysis as illustrated in Fig. 3.8.



Exclusions & Filters

Exclude files

```
*.sln;*.csproj;*.resx;*.vbproj;*.xproj;*.xml;*.md;*.txt;*.json;*.Designer.cs
```

A semicolon separated list of file extensions (see examples)

Exclude content

```
repository-root-folder/glob pattern
```

A semicolon separated list of glob patterns for content you want

Only analyze content under the src & test folders in the backend repository

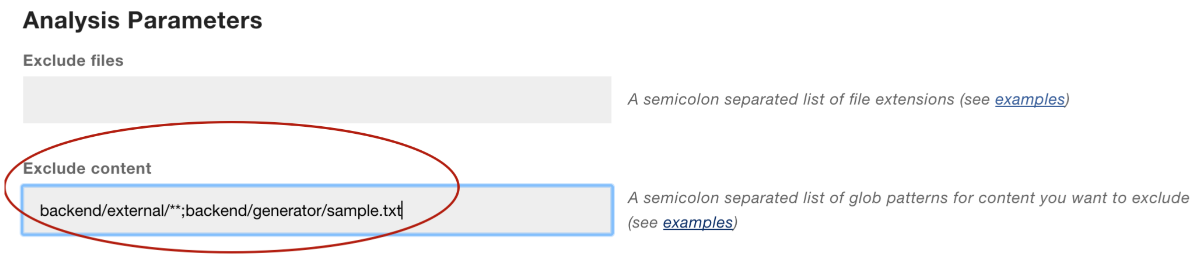
White List content

```
backend/src/**;backend/test/**
```

Specify a semicolon separated list of glob patterns for content you want to include. White listed content is applied before the exclusion patterns above (see examples)

Fig. 3.8: Glob patterns to white list content.

You specify a *glob pattern* to Exclude Content from the analysis as illustrated in Fig. 3.9.



Analysis Parameters

Exclude files

A semicolon separated list of file extensions (see examples)

Exclude content

```
backend/external/**;backend/generator/sample.txt
```

A semicolon separated list of glob patterns for content you want to exclude (see examples)

Fig. 3.9: Glob patterns to exclude content.

The example above will exclude all content under the external folder and the file samples.txt from the generator folder.

Note: You need to specify your exclusion paths using UNIX style path names. That is, use forward slashes as separators. Also note that the paths have to start with the name of your repository root. That is, if your Git repository is located in a folder named backend, as in the example above, you have to prepend that folder name to all your exclusion patterns. The reason for that is due to CodeScene's support for multiple repositories where you have to be explicit about what repository you exclude things from.

There's one exception to the rule that patterns have to specify the repository root. That's the case when you want a pattern to apply across all repositories. For example, let's say that you want to exclude all shell scripts in your test folder. In that case you specify a pattern like ***/test/*.sh* That is, your patterns are allowed to start with a wildcard too.

3.1.7 A Brief Guide to Glob Patterns

Glob patterns let you specify paths- and file names with different wildcards. CodeScene supports the following wildcards:

1. ***: A single asterisk matches any string of characters. Use it to exclude or while list particular files. For example **.h* will exclude all files with extension *h*. You can also use the single asterisk to specify glob patterns that apply to *all* your repositories in a multi repository analysis project. For example, the glob pattern **/version.txt* will match (and possibly exclude) the *version.txt* files at the top level of each of your repositories.

2. ****: The double asterisk matches whole paths/directories. You use the double asterisk to exclude or white list content *independent* of the content's location in your codebase. For example, the pattern *myrepository/**/*.h* will match all files with extension *h* in *any* directory in your repository. You can also use the double asterisk to match exclude or white list whole folders. Let's say we want to exclude all our unit tests from an analysis and that those tests are located in the repository 'coolstuff'. Here's a pattern for that: *coolstuff/test/***.

3. *?*: The question mark matches a single character.

Please note that *all* glob patterns are specified using UNIX style path names. That is, if you're on Windows you do *not* use backslash to separate directory names, but rather the UNIX style forward slash. That is, the directory *SomeRepo\Test* is excluded by specifying *SomeRepo/Test/***.

3.1.8 Specify An Analysis Period

CodeScene lets you specify an *analysis period* as illustrated in Fig. 3.10. That is, how much of your repository history do you want to analyze?

The actual analysis period you select depends on several factors:

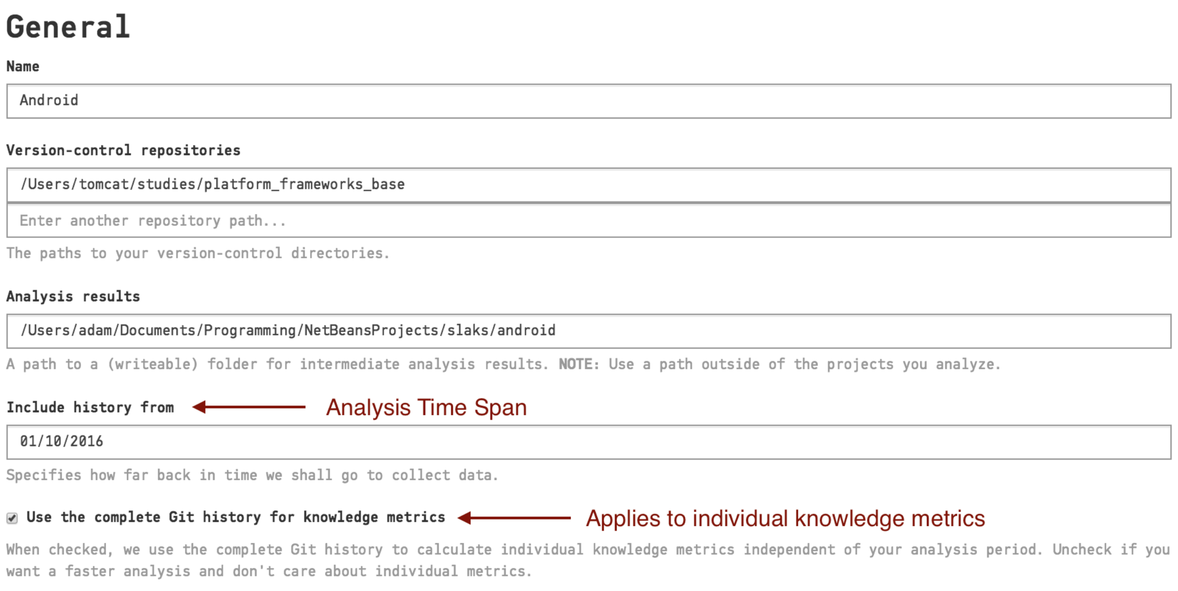
1. *The activity in your project*: Select a short analysis period, like 6 months, in a codebase with a lot of development activity.

2. *The information you want*: If you want an overall view of potential maintenance problems, we recommend that you use a longer analysis period like a couple of years. If, on the other hand, you want to identify recent modifications to the codebase, your analysis period could be as short as the length of your iterations (2-3 weeks).

3. *You have recently re-structured the codebase*: In this case you want to specify an analysis start date after the re-structuring. The rationale is that the history is probably not as useful since you now have a new structure of your system. Use that as the cut-off point.

By default CodeScene will use two different analysis periods depending on the type of information it analyses:

- Technical information and Team information uses the specified date.


General

Name

Android

Version-control repositories

/Users/tomcat/studies/platform_frameworks_base

Enter another repository path...

The paths to your version-control directories.

Analysis results

/Users/adam/Documents/Programming/NetBeansProjects/slaks/android

A path to a (writeable) folder for intermediate analysis results. NOTE: Use a path outside of the projects you analyze.

Include history from ← Analysis Time Span

01/10/2016

Specifies how far back in time we shall go to collect data.

[✓] Use the complete Git history for knowledge metrics ← Applies to individual knowledge metrics

When checked, we use the complete Git history to calculate individual knowledge metrics independent of your analysis period. Uncheck if you want a faster analysis and don't care about individual metrics.

Fig. 3.10: Specify how far back in time you want CodeScene to analyse.

- Individual knowledge metrics use the full history of your repository.

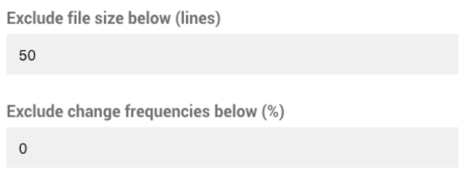
The rationale is that analyses on the level of individual developers, like knowledge maps and knowledge loss, need to take the full history of the codebase into account in order to be accurate. You can disable this behavior and use the specified date for all analyses by unchecking the box "Use the complete Git history for knowledge metrics" (see Fig. 3.10).

Finally, please remember that selecting an analysis time span depends on the questions you have. As such your choice depends on your context and is more of a heuristic than a science. Always start with an analysis of the full history when in doubt.

3.1.9 Visualization Options

CodeScene is capable of analyzing large codebases consisting of millions lines of code. However, the web browser you use to view the results isn't always that performant. In particular, if you have a repository with several thousands of files, the Hotspot and Knowledge visualizations will become slow and painful to navigate.

If you experience that problem, consider to increase the thresholds in the *Visualization* section, shown in Fig. 3.11.


Exclude file size below (lines)

50

Exclude change frequencies below (%)

0

Fig. 3.11: Exclude small files from a visualization.

The first option simply excludes files smaller than your specified size from the visualizations. The second option excludes files that haven't changed more often than the threshold you enter.

The rationale is that in a system of several thousand files, the small ones (1-100 Lines of Code) are probably not the most interesting ones. Thus, these should be safe to exclude.

Note that the visualization algorithm performs some checks to ensure that a hotspot, no matter how small, is included anyway so that you don't miss some important result. Also note that the exclusion only applies to the visualization - the code is still included in the analysis.

3.2 Configure Developers and Teams

Your knowledge maps are based on colors to give you an accessible high-level overview. The system will automatically assign a distinct color to each top-contributor in your codebase on the first analysis. But you still need to assign a color to the other developers if you want to identify them in the visualizations. Similarly, you can define teams of developers and assign each team a color as well.

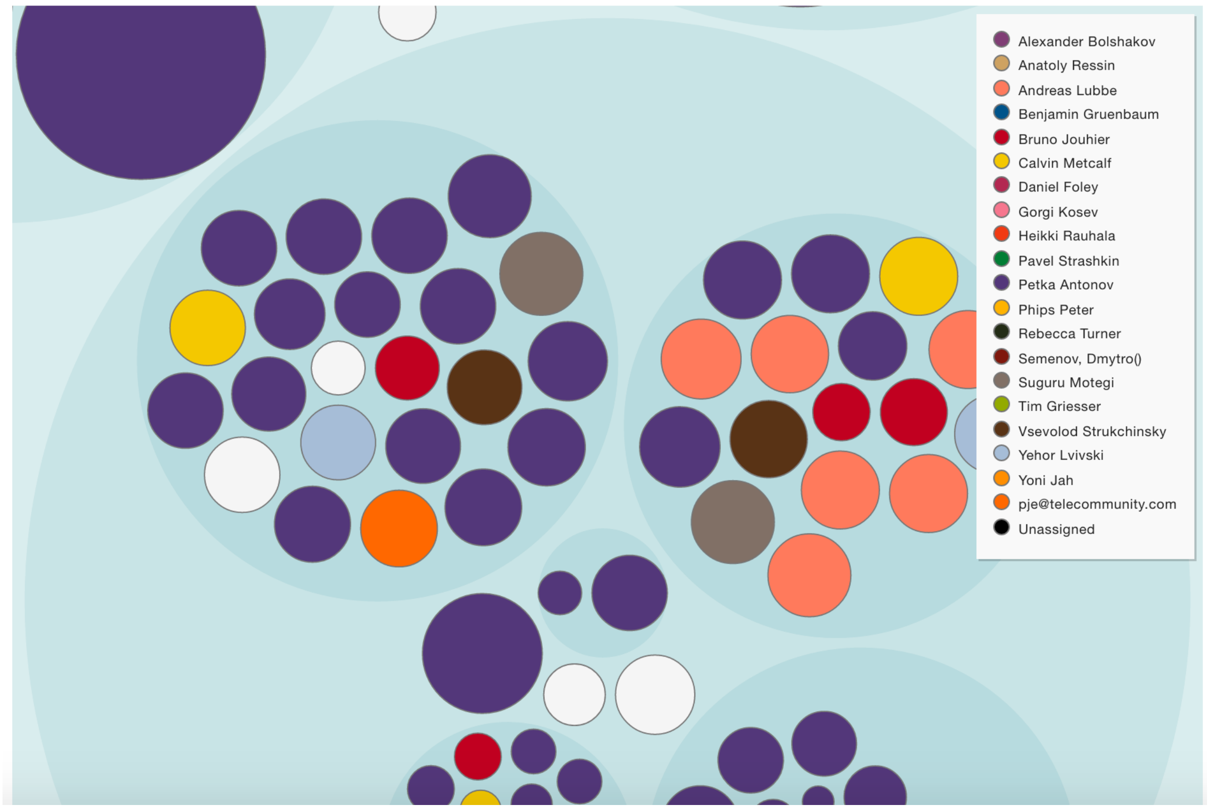


Fig. 3.12: Sample on colored knowledge maps.

The rest of this guide will walk you through the configuration.

3.2.1 Important: Run an Initial Analysis Before You Configure Developers

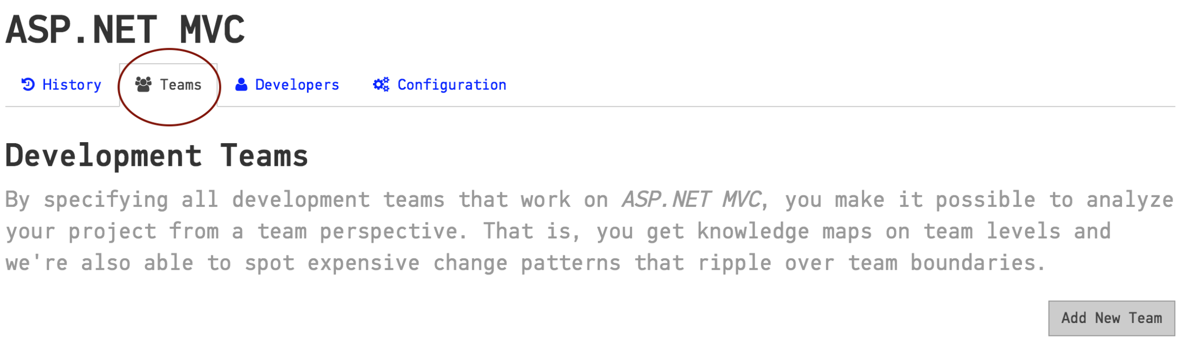
CodeScene mines a list of all contributing developers. Note that this list is mined and updated during each analysis. That means you need to run one initial analysis *before* the tool gives you the option to configure developer properties!

3.2.2 Define Your Development Teams

Click the *Teams* tab in your project configuration to proceed to the teams configuration, as shown in Fig. 3.13.

For each team in your organization, specify the following properties:

- *Team name:* This will be used to identify the team. Later, when you configure developers, you'll assign them to the team names you chose here.
- *Team Color:* The team color is used in the visualization of knowledge distribution on team level.

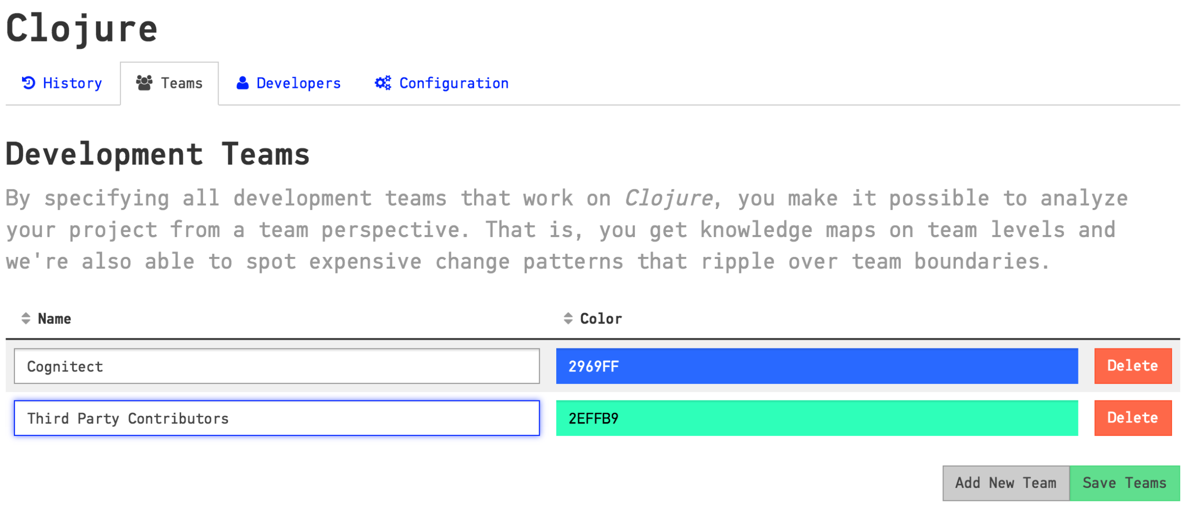


Development Teams

By specifying all development teams that work on *ASP.NET MVC*, you make it possible to analyze your project from a team perspective. That is, you get knowledge maps on team levels and we're also able to spot expensive change patterns that ripple over team boundaries.

Fig. 3.13: Configure teams for a project in the Teams tab.

Add a team for each team in the organization that works on your codebase (see Fig. ??).



Development Teams

By specifying all development teams that work on *Clojure*, you make it possible to analyze your project from a team perspective. That is, you get knowledge maps on team levels and we're also able to spot expensive change patterns that ripple over team boundaries.

Name	Color	
Cognitect	2969FF	Delete
Third Party Contributors	2EFFB9	Delete

Tip: Some organizations just use one development team. In that case, introduce virtual teams that depend upon the responsibilities of the different developers. For example, you might want to define a Feature team, a Maintenance team and an Infrastructure team. Using this strategy, you'd be able to identify code at risk for incompatible parallel changes since different forces lead to the changes.

Even Open-Source Software Has Teams

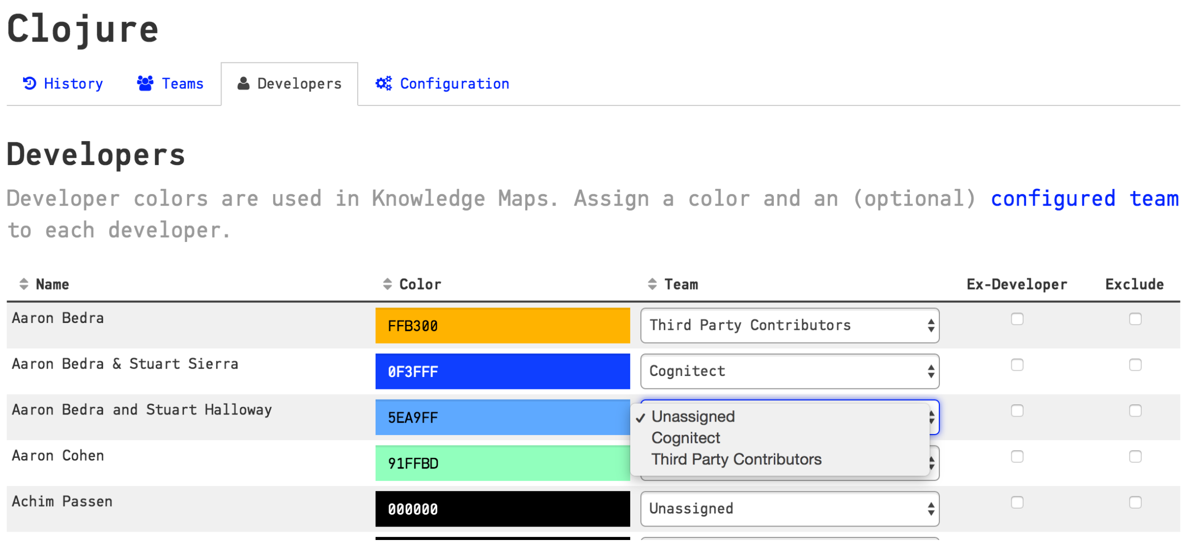
The team definition is straightforward if you analyze a codebase that's owned by a traditional organization; Just use the information from your organizational chart. However, we find it interesting to apply teams to open-source codebases as well.

So if you happen to analyze an open-source project, consider introducing the following teams to get additional social information:

- Define a teams for the organization that owns the code. For example, if you analyze the Clojure codebase, you'd define *Cognitect* as one team. If you analyze one of Microsoft's open-source codebases, you'd use *Microsoft* as one team.
- Define a team for third party developers that contribute to the codebase
- Consider defining a team of the core maintainers too.

3.2.3 Configure Developer Properties

The developer properties are a bit more tricky than the team configuration, so please let us walk you through them one by one as illustrated in Fig. 3.14.



Clojure

Developers

Developer colors are used in Knowledge Maps. Assign a color and an (optional) configured team to each developer.

Name	Color	Team	Ex-Developer	Exclude
Aaron Bedra	FFB300	Third Party Contributors	☐	☐
Aaron Bedra & Stuart Sierra	0F3FFF	Cognitect	☐	☐
Aaron Bedra and Stuart Halloway	5EA9FF	Unassigned	☐	☐
Aaron Cohen	91FFBD		☐	☐
Achim Passen	000000	Unassigned	☐	☐

Fig. 3.14: Specify organizational information for each developer.

CodeScene automatically updates the list of contributing developers; If a new developer starts to contribute code, they'll be present in the list and the tool lets you configure their properties.

Here are the properties you need to specify:

1. *Color:* The color is used on the knowledge maps to uniquely identify a developer. Try to assign the top contributors as distinct colors as possible.
2. *Active/Ex-Developer:* By default, all developers are considered active. If some of them leaves your project, mark them as *Ex-Developers* and CodeScene will include them in the *Knowledge Loss Analysis*.
3. *Team:* The third column lets you assign the developer to a team. That will include them in the *Team Knowledge Distribution Analysis*.
4. *Exclude author from analysis:* If you check this option, the author will be excluded from *all* social analyses (although their contributions will still be included in the technical analyses like Hotspots and Code Churn). This is an option you use in case you have roles like System Integrators that only merge code, but never actually make their own contributions.

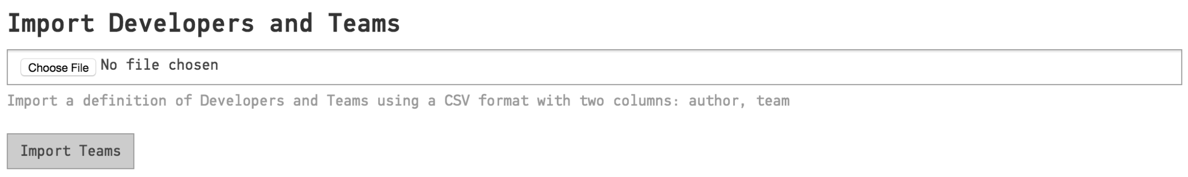
Once you've defined all developer properties you just need to run a new analysis and you'll get a smorgasbord of interesting social analysis results.

3.2.4 Import a Definition of Development Teams

It may well be impractical to configure each team and developer via the UI, particular for large organizations. That's why CodeScene supports import of the organizational chart.

You find the import functionality in the Team configuration:

The input file specifies your organization. The file format has to be a CSV with two columns: author and team.



Import Developers and Teams

Choose File No file chosen

Import a definition of Developers and Teams using a CSV format with two columns: author, team

Import Teams

Fig. 3.15: Importing developer information by uploading a CSV file.

3.3 Hotspot Metrics

You've already learned about Hotspots in *Hotspots* (page 12). In this part of the guide you'll learn more about the metrics behind the hotspot criteria and the rare instances of when you want to specify an alternative algorithm for hotspots.

3.3.1 Understand the possible Bias in Commit Styles

By default, CodeScene uses the commit frequency of each file as the Hotspot criteria; The more changes you've done to a file, the higher its change frequency. This default criteria is supported by several findings from academic research; change alone is the single most important metric when it comes to quality issues in code.

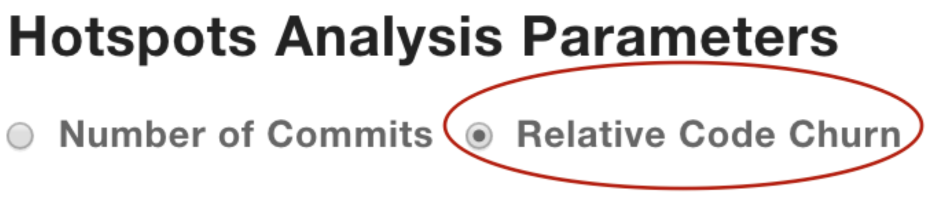
However, there are some rare cases when this metric becomes biased. One reason is large individual differences in commit style. Another possible source of bias is for semi-automated code updates like refactoring via a tool. If you rename a type, all other code that references that type get updated as well. This may even include hyper linked code comments. Since it's a minor change in terms of both programmer effort and risk, one could argue that such changes are less important.

It's important to point out that even when you experience any of the situations above you may still be fine. In a large system, such differences tend to even out. However, if you suspect that the impact is large enough to bias your data, please consider using *Relative Code Churn* as Hotspot criteria instead of the number of commits. Let's see how you can configure that alternative.

3.3.2 Enable Relative Code Churn

Relative Code Churn is an alternative hotspot metric that calculates the amount of change in each file in terms of Lines of Code. It's a relative metric since the churn is weighted against the total size of the code in each file.

You enable Relative Code Churn in your Project Configuration:



Hotspots Analysis Parameters

Number of Commits [✓ Relative Code Churn]

Fig. 3.16: You can switch to "Relative Code Churn" in the project configuration.

From now on, all Hotspots are calculated based on the amount of change to their lines of code.

3.3.3 The Pros and Cons of Relative Code Churn

The major drawback of using churn measures to predict Hotspots, is that small files that change often get a very high churn rate. CodeScene automatically adjust for this scenario, but it's still something you want to be aware of.

Relative Code Churn has other uses besides providing an attractive alternative in case of biased commit styles. For example, you'll find that old, now stable files fall much faster on the Hotspot ranking ones the development efforts shift to other parts of the codebase. With number of commits, a historic Hotspot that have since been refactored, may still stick around the top ranking due to its troubled past.

Finally, you'll probably find that there's a high correlation between the two Hotspot metrics. In that case, prefer the number of commits metric since it's easier to reason about.

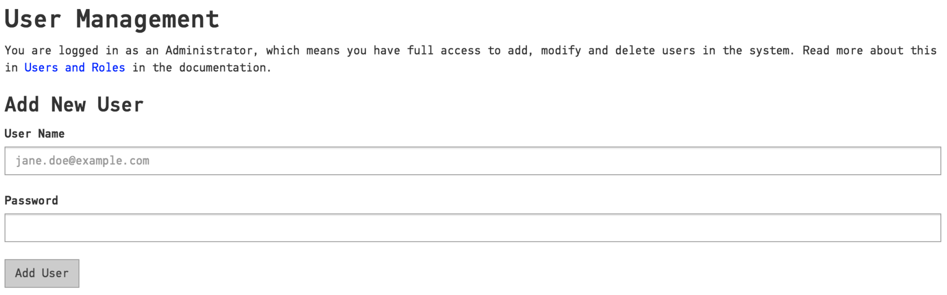
3.4 Users and Roles

CodeScene lets you create users and grant them various levels of access depending on their roles.

3.4.1 Adding Users

When logging in with your Licensee Name and Product Key you receive full administrative privileges. Some tasks require these special privileges, such as deleting projects and managing the global configuration. We recommend using the administrator login only for such tasks, and creating user accounts with restricted access for regular work.

By clicking the "Configuration" tab in the top navigation bar you get to the global configuration page. If you are logged in as the administrator, you should see the Users configuration, as in Fig. 3.17.



User Management

You are logged in as an Administrator, which means you have full access to add, modify and delete users in the system. Read more about this in Users and Roles in the documentation.

Add New User

User Name

jane.doe@example.com

Password

Add User

Fig. 3.17: In the global configuration you can add new users to the system.

Enter the user name and password, and click "Add User" to finish. The password can be changed later if needed, either by the administrator or the users themselves.

3.4.2 Assigning Roles

The system comes preconfigured with a number of roles. You can assign roles to the users in your system to grant them specific access.

Technical Technical analyses only.

Developer Technical, architectural and social analyses.

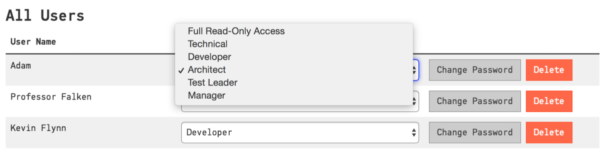
Architect Technical, architectural and social analyses.

Test Leader Hotspot and knowledge map analyses.

Manager Technical quality guide and social analyses.

Full Read-only Access Access to all analyses, but cannot perform any actions. Can be used for guest access.

In the table of existing users you can see the currently assigned roles. Click on the *Role* select box, as shown in Fig. 3.18, to change the assigned role of a user.



All Users

User Name			
Adam		Change Password	Delete
Professor Falken		Change Password	Delete
Kevin Flynn	Developer	Change Password	Delete

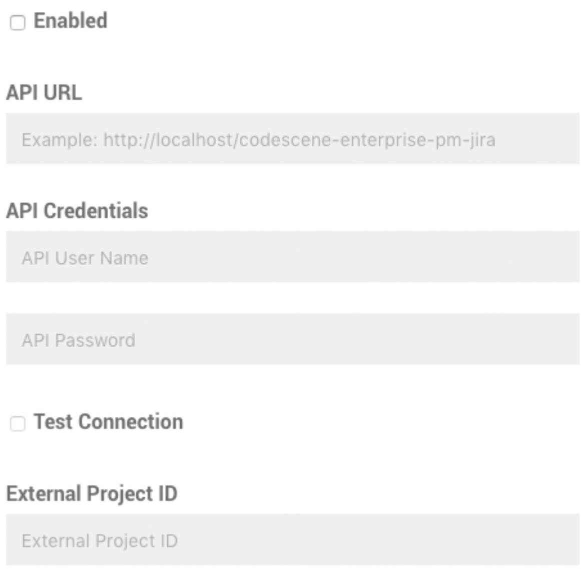
Fig. 3.18: By clicking the Role select box you can change the assigned role of a user.

3.5 Project Management Integration

CodeScene supports integration with project management (PM) systems, like JIRA. Issues in the PM system are mapped to the corresponding commits in the version control system.

3.5.1 Repository Configuration

By default, PM integration is disabled (see Fig. 3.19). Enable by checking the 'Enabled' checkbox.



☐ Enabled

API URL

Example: http://localhost/codescene-enterprise-pm-jira

API Credentials

API User Name

API Password

☐ Test Connection

External Project ID

External Project ID

Fig. 3.19: Check 'Enabled' to enable the project management integration.

Enabling the integration lets you edit the remaining fields (see Fig. 3.20):

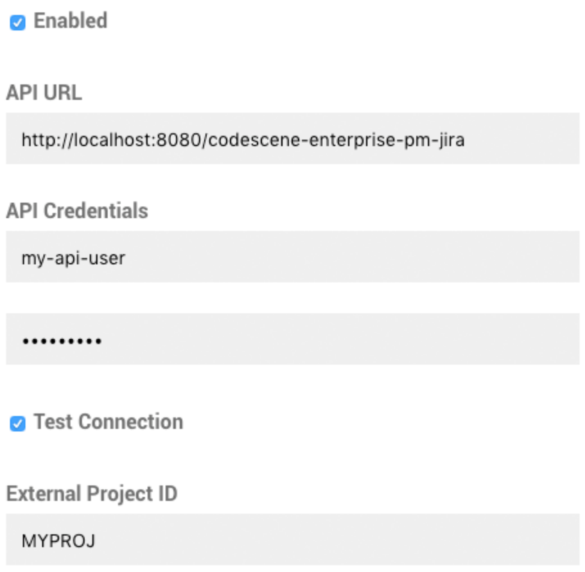
API URL The base URL of the PM integration service. If you have deployed the JIRA integration in Tomcat, the URL will likely be *http://localhost:8080/codescene-enterprise-pm-jira*.

API Credentials The credentials needed to access the PM integration service. Note that these are the credentials that are configured in the PM integration service.

Test Connection Try connecting using the specified API URL and credentials, and check the status of the PM API, before saving the configuration. Use this option to verify the connection before running an analysis.

External Project ID The project identifier in the external system. If the external system is JIRA, this field should contain the *JIRA project key*. For example, if issues are named *MYPROJ-123*, the project key (and thus the external project ID) is *MYPROJ*.

You can add multiple JIRA projects here by separating them with a semicolon, *;* as shown in Fig. 3.21


✓ Enabled

API URL

http://localhost:8080/codescene-enterprise-pm-jira

API Credentials

my-api-user

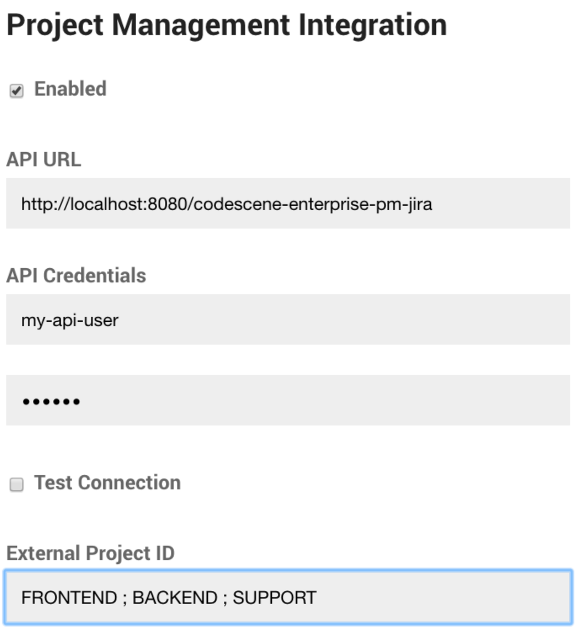
•••••••••

✓ Test Connection

External Project ID

MYPROJ

Fig. 3.20: A configuration sample for project management integration.


Project Management Integration

✓ Enabled

API URL

http://localhost:8080/codescene-enterprise-pm-jira

API Credentials

my-api-user

••••••

☐ Test Connection

External Project ID

FRONTEND ; BACKEND ; SUPPORT

Fig. 3.21: A configuration sample for integration with multiple projects.

3.5.2 Ticket ID Configuration

Each item from the PM integration has an ID that needs to match the *Ticket IDs* in CodeScene. For example, when integrating with JIRA, the mapping needs to extract the ID part from the JIRA issue key. In addition to mapping item IDs from the PM system, the ticket IDs need to be extracted from the VCS logs, which is called *Ticket ID Mapping*. *Measure Temporal Coupling across Multiple Repositories* (page 61) explains Ticket ID Mapping in greater detail. Fig. 3.22 illustrates how both mappings extracts IDs with the same format.

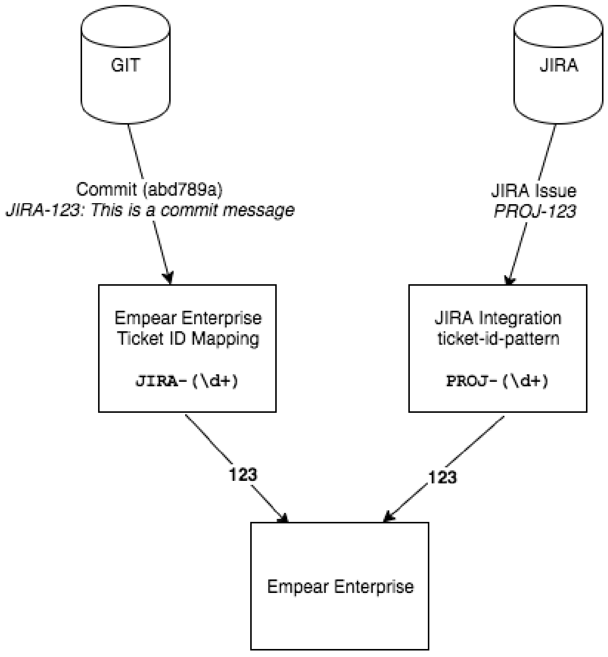


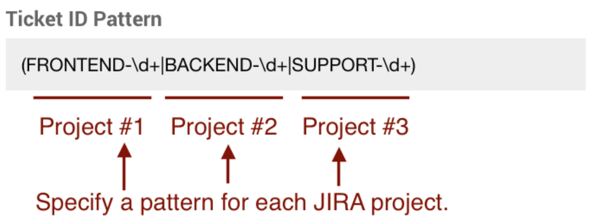
Fig. 3.22: Ticket IDs are extracted from the VCS logs using Ticket ID Mapping, and Project Management Item IDs are mapped from JIRA issue keys using a configured pattern in the JIRA integration service.

Ticket ID Configuration for Multiple JIRA Projects

Please note that in case you integrate with multiple JIRA projects, you may have to use a different Ticket ID configuration in case the ID's may overlap.

For example, let's say you integrate with three projects. Each project will have a JIRA ID like *FRONTEND-123*, *BACKEND-765* and so on. In this case you want to use the whole JIRA ID as a Ticket ID to ensure that they are unique. In addition, you need to specify a regular expression that will match *all* your possible JIRA ID ranges

Fig. 3.23 shows an example on such a configuration.


Ticket ID Pattern

(FRONTEND-\d+|BACKEND-\d+|SUPPORT-\d+)

Fig. 3.23: Ticket ID specification that matches items from multiple JIRA projects.