CODESCENE

Enterprise Edition

`3.3.11`

October 11, 2019

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 4) 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.
- *Hands On Behavioral Code Analysis: CodeScene Use Cases* (page 19) shows how to get the most out of CodeScene by learning its use cases and how to apply and integrate the analysis information in your daily work.
- *Integrations* (page 27) explains how you integrate CodeScene into CI/CD and Jira to enable quality gates and to measure lead times in your development process.
- *Guides* (page 36) walk you through specific features and aspects of the tool, focusing on how you can use them to achieve certain goals.
- *Configuration* (page 146) explains how you configure projects to get the best possible analysis results.
- usage/language-support covers CodeScene's language level support.

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 2.14*. 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.

1.1.3 Persistent Authentication Sessions

CodeScene stores user sessions in memory which means that users have to log in every time the application process is restarted.

Sometimes, it can be useful to avoid re-authentication, especially when you run CodeScene under a system user for monitoring purposes (Dashboard view).

In that case, you can turn on "persistent" authentication via browser cookies by setting *AUTH_SESSION__ENCRYPTION_KEY* environment variable. Encryption keys must be 16 characters long! User session stays valid for 14 days. If you want to change this interval, you do so via *AUTH_SESSION__MAX_AGE_SECONDS* environment variable.

```
# by setting encryption key you make users sessions persistent across restarts
export AUTH_SESSION__ENCRYPTION_KEY="1234567890abcdef"

# optionally, you can change max-age too
export AUTH_SESSION__MAX_AGE_SECONDS=120
```

1.1.4 Set up a Proxy Server

If CodeScene is running behind a proxy server, you might need to specify the proper configuration.

Whenever you're trying to login or activate the license, Codescene contacts the license server to check if the license is valid and to update license limitations.

Without the correct proxy configuration, CodeScene won't be able to check the license and update the license limitations in case they were changed on the server and will show an error message (unless you're in offline mode - more on that later).

The user can provide the proxy configuration when logging in. If the license check is successful and the user is admin, the proxy configuration will be automatically saved in the global config for future connections.

Proxy Without Authentication

If you use a proxy server that does not require authentication, CodeScene might be able to automatically detect the configuration based on your operating system settings. You can always check current the proxy configuration in *Configuration -> License -> Proxy Server*.

Proxy with Basic authentication

If your proxy server is configured to use Basic authentication, you need to provide a username and password. Please, fill in the 'User' and 'Password' fields in *Configuration -> License -> Proxy Server*.

Proxy with Kerberos Authentication

CodeScene supports proxy servers with Kerberos authentication.

As long as you have a valid TGT ticket in your system's Credentials Cache, CodeScene should be able to authenticate with your Proxy Server.

This is usually done with *kinit* command:

```
kinit <principal_name>
```

However, TGT tickets have limited validity (usually 24 hours). If you aren't able to refresh them automatically, you need to specify the username (principal) and password in *Configuration -> License -> Proxy Server*.

If you don't want to store username/password in CodeScene you can also create a keytab file and specify it in *login.conf* in CodeScene root folder as follows (make sure to put proper *principal* and *keyTab* file path):

```
com.sun.security.jgss.initiate {
  com.sun.security.auth.module.Krb5LoginModule required client=TRUE useTicketCache=true␣
↪doNotPrompt=false refreshKrb5Config=true
  principal=codescene useKeyTab=true keyTab=codescene.keytab;
};
com.sun.security.jgss.accept {
  com.sun.security.auth.module.Krb5LoginModule required client=TRUE useTicketCache=true␣
↪doNotPrompt=false refreshKrb5Config=true
  principal=codescene useKeyTab=true keyTab=codescene.keytab;
};
```

You can learn more about using Kerberos in Java applications here: Use of Java GSS-API for Secure Message Exchanges Without JAAS Programming:.

Other Authentication Mechanisms

If you're using another proxy authentication mechanism and you're not able to make it work with CodeScene, please let us know. We'll do our best to add support for this authentication mechanism to CodeScene.

Offline Mode

If you are unable to provide a proper proxy configuration, you don't want to let Codescene reach the Internet, or you simply don't have an Internet connection for a limited period of time, you still may run CodeScene in *offline mode*.

If CodeScene is unable to activate its license or verify the user's login, it will show an error message with a checkbox for activating offline mode. Administrators can also turn on offline mode globally in Configuration.

Offline mode is limited to the period of the current subscription term (billing period). When your current billing period ends, CodeScene needs Internet access to verify your license. If you need to run in offline mode for an extended period of time, you will have to pay for the whole period in advance.

Please note that offline mode should be regarded as an exceptional use case, either for emergencies or situations with specific needs. To ensure a smooth experience, users are encouraged to provide proper configuration. Please contact us if you need more help with network/proxy configuration.

Connection Timeouts

By default, CodeScene uses a 5000 ms timeout for both connection timeout and socket timeout. You can customize these settings with the *LICENSE_CHECK_CONN_TIMEOUT* and *LICENSE_CHECK_SO_TIMEOUT* environment variables. This can be useful if you're running CodeScene in a high-latency environment or in permanent offline mode.

You can also enforce a hard timeout on the whole duration of license check request with *LICENSE_CHECK_TOTAL_TIMEOUT* environment variable. If you don't specify total timeout computed value *1.5 * (connection timeout + socket timeout)* is used as a default. Total timeout gives you a complete control over the license check duration. You cannot achieve this using just connection timeout and/or socket timeout. The main difference is in inability to control DNS resolution time using either connection or socket timeout.

1.2 Installation

CodeScene is quite flexible in that you can run it in any environment that meets the requirements specified in *Configure your Environment* (page 4).

In general, there are two popular options for hosting CodeScene:

- *CodeScene inside a Docker container*: We have an example on a docker file here, including an example on how to configure SSL. With this option, you could also deploy CodeScene to a private cloud (e.g. aws). In this case, you'd use the standalone CodeScene application.
- *CodeScene hosted by Tomcat*: Tomcat might be an option if you already host other applications on Tomcat.

1.2.1 Run the Standalone CodeScene Application

The easiest way to get CodeScene 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`.

Once you've launched the *codescene.standalone.jar* you just point your web browser to *localhost:3003* to access CodeScene.

Define the Root Paths for Persistent Data

CodeScene creates a local database for the analysis configurations. By default the database file `resources/caacs_enterprise.db.mv.db` is created in the working folder (that is, the directory where you run CodeScene).

You can override the default and provide a custom path through the environment variable `CODESCENE_DB_PATH`. Note that you need to specify a complete file name. As an example, if you specify `/User/Services/CodeScene/configuration`, CodeScene will create a persistent database file named `/User/Services/CodeScene/configuration.mv.db`.

CodeScene will also need access to a writeable file system where it can store analysis results and cloned Git remotes. In some environments (e.g. Docker) you might want to constrain those paths so that users don't specify paths to a non-persistent storage. You do this by two *optional* environment variables:

1. `CODESCENE_ANALYSIS_RESULTS_ROOT` : Specifies the root path to where all analysis results will be written. CodeScene auto-generates a folder for each analysis project.
2. `CODESCENE_CLONED_REPOSITORIES_ROOT` : Specifies the root path to where Git remotes will be cloned locally. CodeScene auto-generates a folder for each analysis project.

The advantage of specifying these two optional environment variables is that the user won't have to deal with configuring the result paths – it's automated – and analysis results are always stored to a known partition.

Configure the available Memory

RAM is a critical resource for CodeScene. In most cases 4G of RAM is more than enough, but if your codebase has large files (we mean really large, like +30,000 lines of code) you may need more memory to run the X-Ray analyses.

Note that Java's virtual machine has a system dependent maximum that is typically lower than the total RAM available. That means you need to specify a higher threshold yourself when starting CodeScene. You do that by providing the `-Xmx` flag to java.

Here's an example that shows how to allocate 10 gigabyte of RAM for CodeScene:

```
java -Xmx10G -jar codescene.standalone.jar
```

Note that the order of the arguments matter in this case.

Avoid missing stack traces

In some situations, the JVM skips stack trace generation and you won't get full stack strace details which may make troubleshooting more difficult. You'll find a message indicating this happened in the logs, e.g.:

```
java.lang.ClassCastException
Stack trace of root exception is empty; this is likely due to a JVM optimization that can be␣
↪disabled with -XX:-OmitStackTraceInFastThrow.
```

You can disable this optimization and make sure stack traces are always visible by using the *-XX:-OmitStackTraceInFastThrow* flag when starting the application:

```
java -XX:-OmitStackTraceInFastThrow -jar codescene.standalone.jar
```

Optional: Run CodeScene in Kubernetes

Finally – and entirely optional – the CodeScene standalone *JAR* could also be run on Kubernetes. This might be an option if your organization already uses Kubernetes as a container management tool.

Follow the instructions here to setup CodeScene on Kubernetes.

1.2.2 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.

DB username and password

Optionally, you can specify a custom username and password to access the database. By default, CodeScene uses the 'sa' user with an empty password.

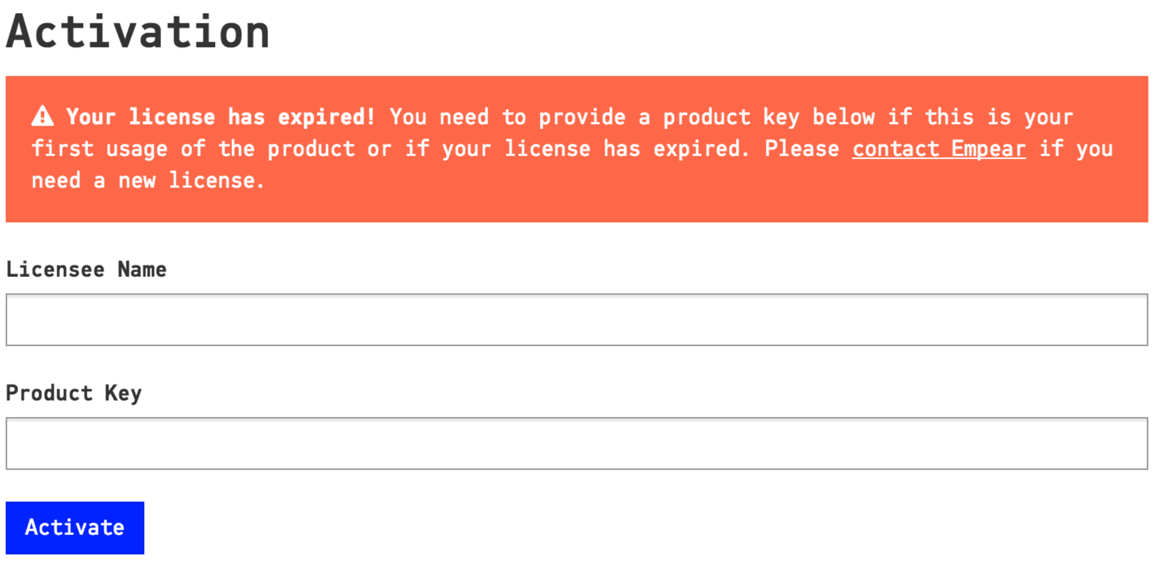
Add `empear.dbuser` and `empear.dbpassword` to the Context environment properties to customize DB username/password.

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).

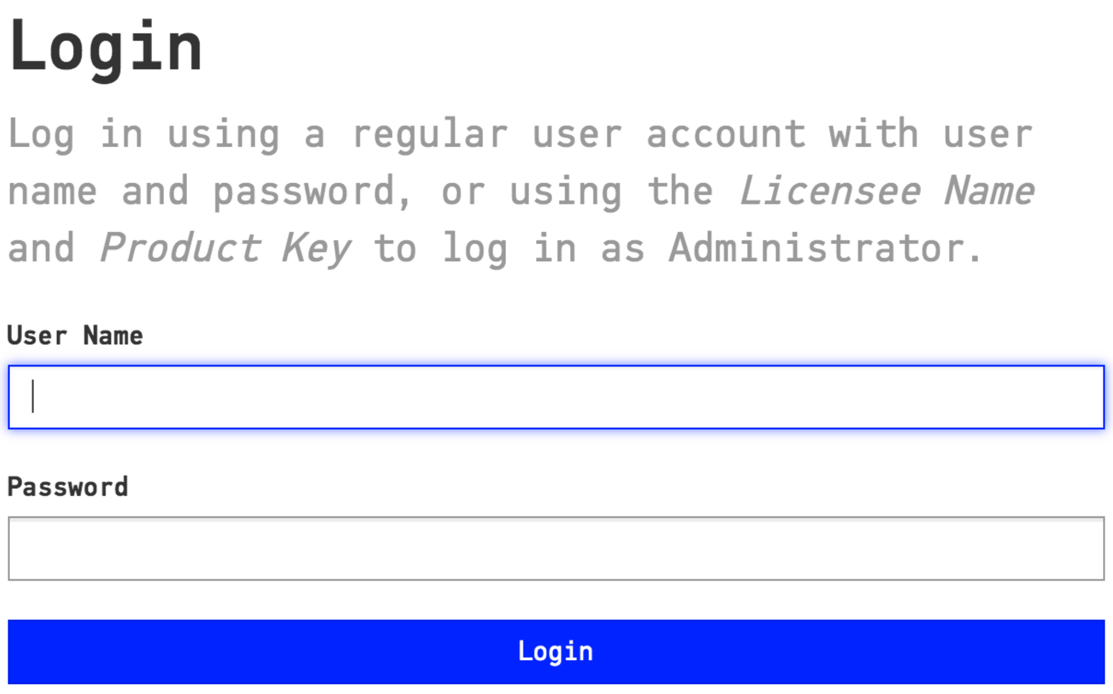
The first time you login, you use *the same* credentials to login as you used to activate the application. That is, give your CodeScene Username as User Name and your CodeScene License Key as Password.

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 166) describes this in greater detail.



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.

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

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 "New Project" button (see Fig. 1.3).

Once you click the "Create New Project" button you are prompted with six choices (see Fig. 1.4):

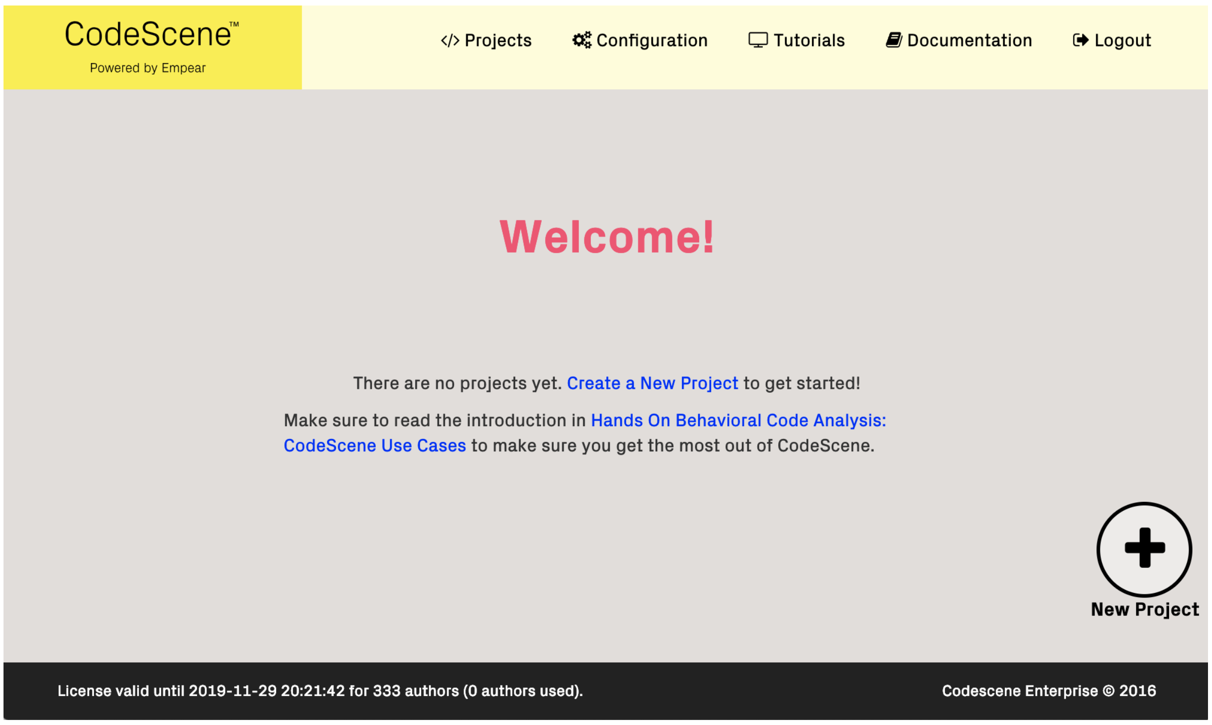
1. *Specify Paths* if you plan to analyze just one or two repositories and enter the paths manually.
2. *Scan Directory* to auto-import multiple repositories into your analysis project.
3. *Specify Remotes* let you specify Git URLs (e.g. to GitHub) and CodeScene automatically clones the repositories.
4. *Clone Existing* to copy an existing analysis configuration into a new configuration. This is useful if you want to provide different analysis views, for example for variying time periods, for the same codebase.
5. Use *Google Repo* to let the Repo tool manage your repositories based on a remote manifest.
6. *Import Configuration* to create a new project based on a previously exported configuration.

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 146) 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.

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



Welcome!

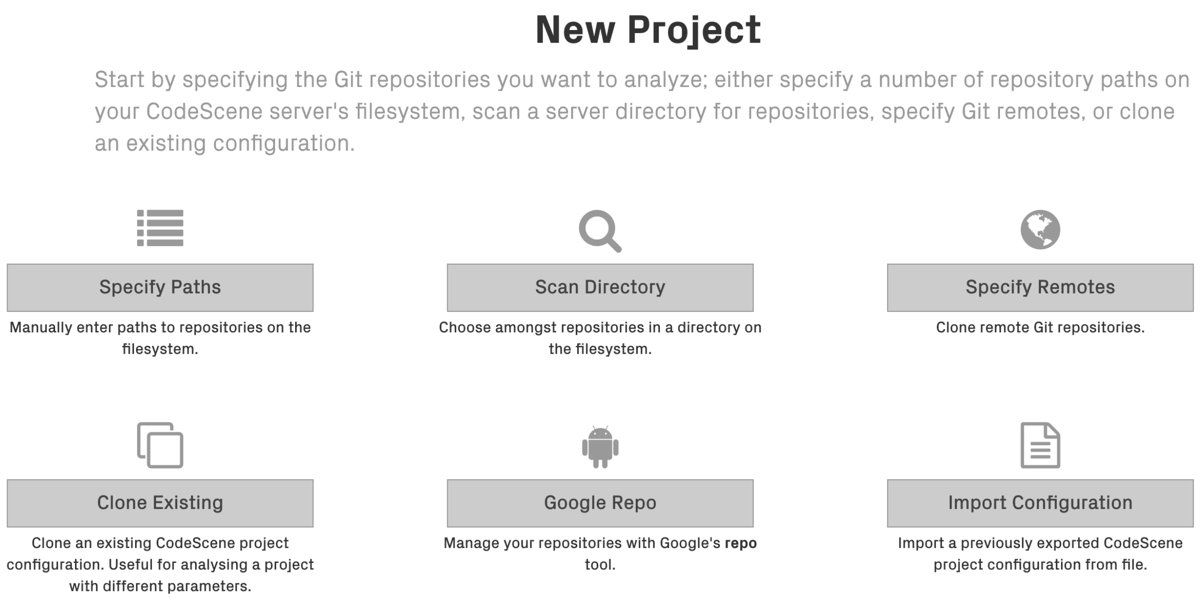
There are no projects yet. Create a New Project to get started!

Make sure to read the introduction in Hands On Behavioral Code Analysis: CodeScene Use Cases to make sure you get the most out of CodeScene.

License valid until 2019-11-29 20:21:42 for 333 authors (0 authors used).

Codescene Enterprise © 2016

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



New Project

Start by specifying the Git repositories you want to analyze; either specify a number of repository paths on your CodeScene server's filesystem, scan a server directory for repositories, specify Git remotes, or clone an existing configuration.

Specify Paths
Manually enter paths to repositories on the filesystem.

Scan Directory
Choose amongst repositories in a directory on the filesystem.

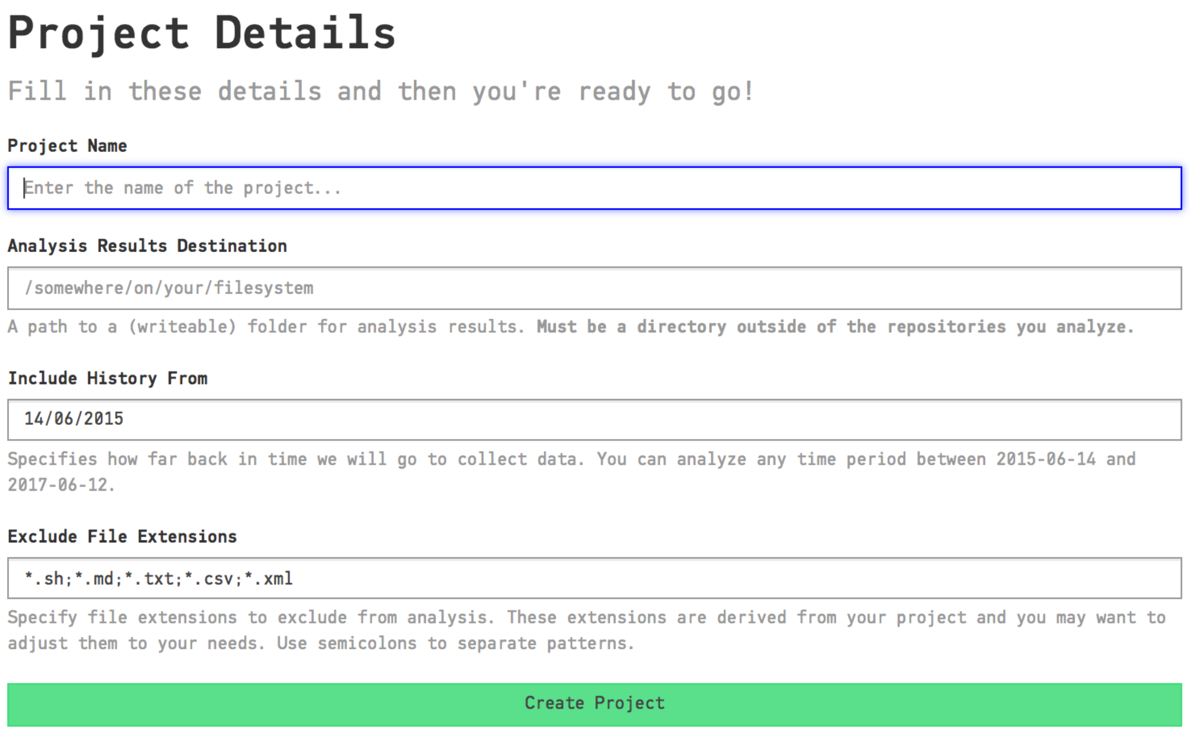
Specify Remotes
Clone remote Git repositories.

Clone Existing
Clone an existing CodeScene project configuration. Useful for analysing a project with different parameters.

Google Repo
Manage your repositories with Google's **repo** tool.

Import Configuration
Import a previously exported CodeScene project configuration from file.

Fig. 1.4: Specify the paths to the Git repositories you want to analyze.



Project Details

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

Project Name

Enter the name 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

14/06/2015

Specifies how far back in time we will go to collect data. You can analyze any time period between 2015-06-14 and 2017-06-12.

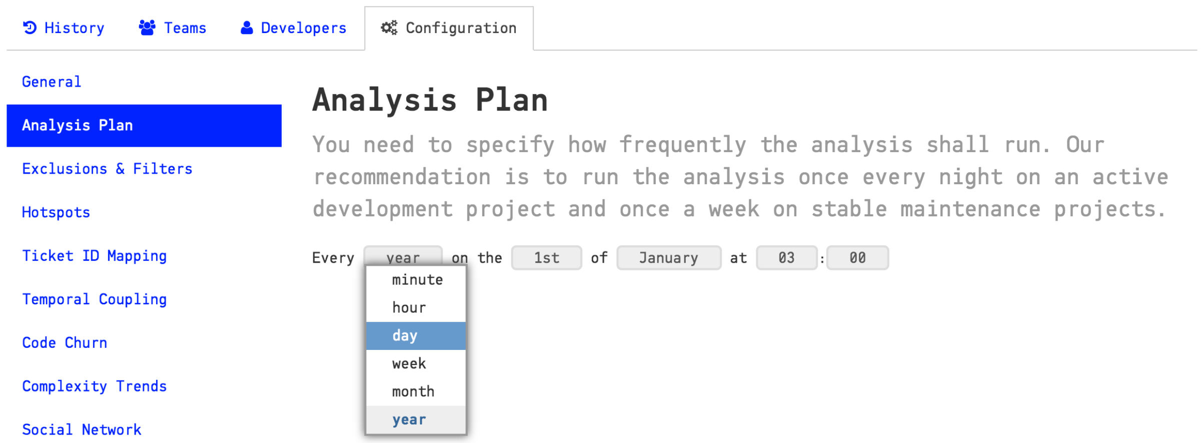
Exclude File Extensions

.sh;.md;*.txt;*.csv;*.xml

Specify file extensions to exclude from analysis. These extensions are derived from your project and you may want to adjust them to your needs. Use semicolons to separate patterns.

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

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.



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.

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.

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.

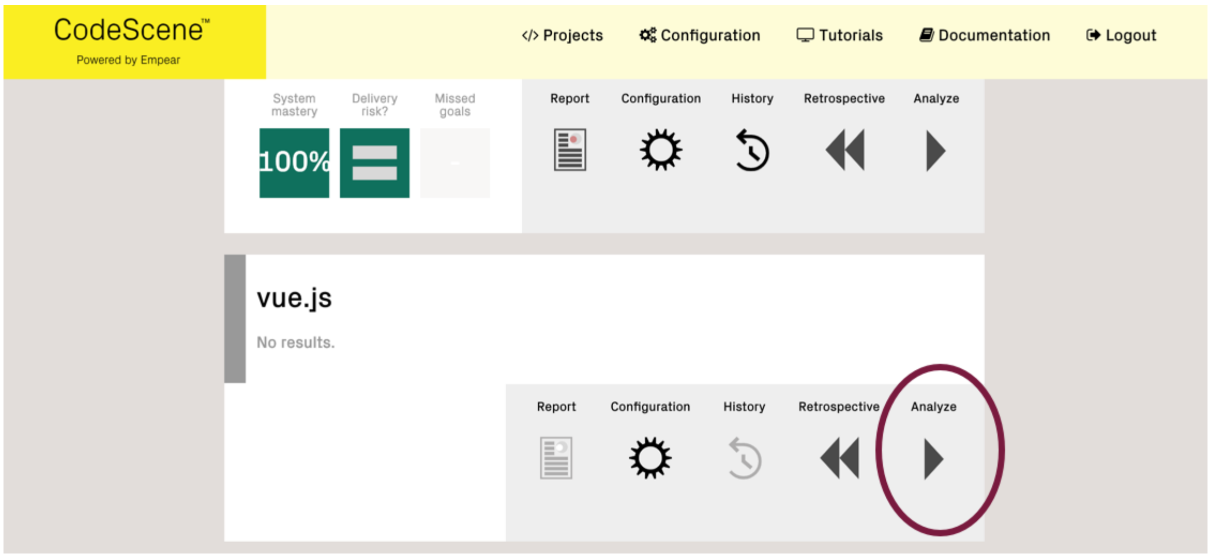
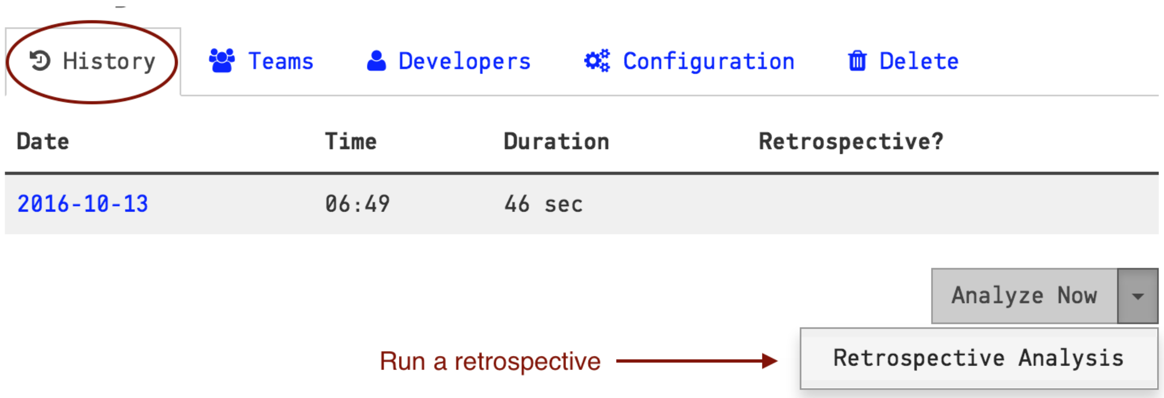


Fig. 1.7: Press the Run button to force an analysis.

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.



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.

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 tile representing your project to inspect your analysis results.
3. Click on the "CODESCENE" logo in the top-left corner to return to the main screen, should you ever get lost.

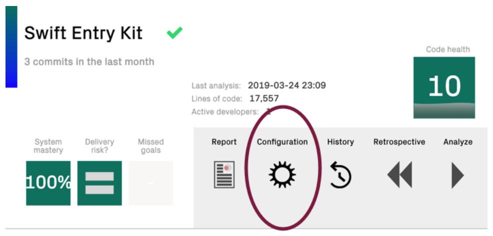


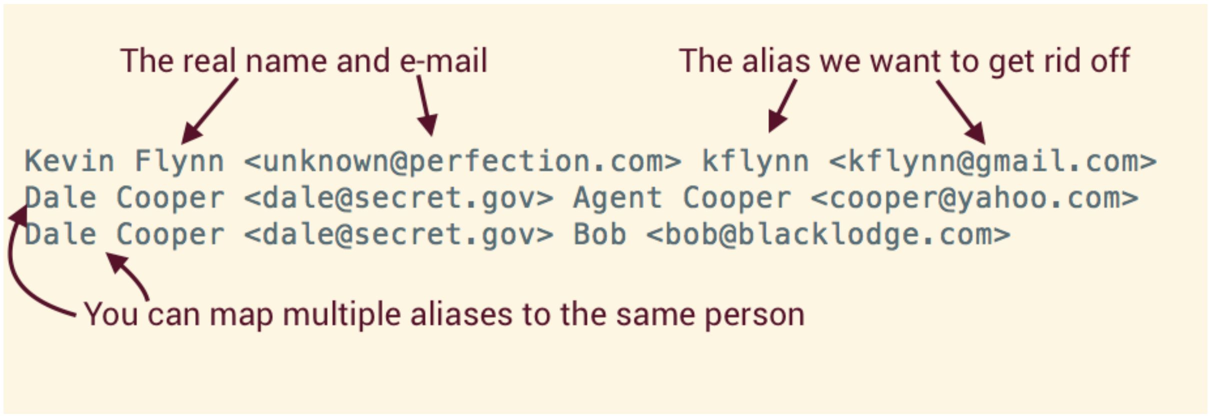
Fig. 1.9: The cogs button in the project tile takes you to the project details and configuration.

1.4 Resolve Developer Aliases

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.

CodeScene provides two solutions to this problem. The simplest is to use the Developer identity mapping interface. (See *Developers and their Aliases: Mapping Version-Control Names to People* (page 161).)

CodeScene also supports Git mailmaps and will automatically use them if they are present. To use mailmaps, add the `.mailmap` file to the root of your repository. It specifies a mapping from multiple aliases to one for each developer as shown in Fig. 1.10.


The real name and e-mail
The alias we want to get rid off
You can map multiple aliases to the same person

```
Kevin Flynn <unknown@perfection.com> kflynn <kflynn@gmail.com>
Dale Cooper <dale@secret.gov> Agent Cooper <cooper@yahoo.com>
Dale Cooper <dale@secret.gov> Bob <bob@blacklodge.com>
```

Fig. 1.10: Resolve aliases through a mailmap.

Note that mailmaps operate at a lower level, so changes in mailmaps will not be visible in the Developer identity mapping interface.

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 encryption. We recommend Nginx as the reverse proxy. The Nginx website provides documentation on configuring Nginx for HTTPS.

Here is a brief example of an Nginx proxy configuration:

```
http {

  server {
    listen 80;
    server_name codescene.example.com;
    location / {
      return 301 https://$host$request_uri;
    }
```

(continues on next page)

(continued from previous page)

```
}

server {
  listen 443 ssl;
  server_name codescene.example.com;

  ssl_certificate /etc/ssl/certs/nginx-selfsigned.crt;
  ssl_certificate_key /etc/ssl/private/nginx-selfsigned.key;

  location /  {
    proxy_pass http://localhost:3003;
    proxy_redirect http:// $scheme://;
    proxy_set_header Host $host;
    proxy_set_header X-Real-IP $remote_addr;
    proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
    proxy_set_header X-Forwarded-Proto $scheme;
  }
}
```

The proxy_redirect used above will rewrite all HTTP redirects from upstream to the current scheme, ie HTTPS. The browser will then receive the correct scheme directly, avoiding unnecessary round-trips.

There is also a Docker-based sample project that provides CodeScene wrapped by an Nginx reverse proxy with a self-signed certificate. It composes a CodeScene container and an Nginx container using a small *docker-compose.yml* file.

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. 1.11). Present it on a TV or a big screen in the office and share the automatic updates with your team.

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. 1.12). 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 166).

1.6.2 Supervise your Feature Branches

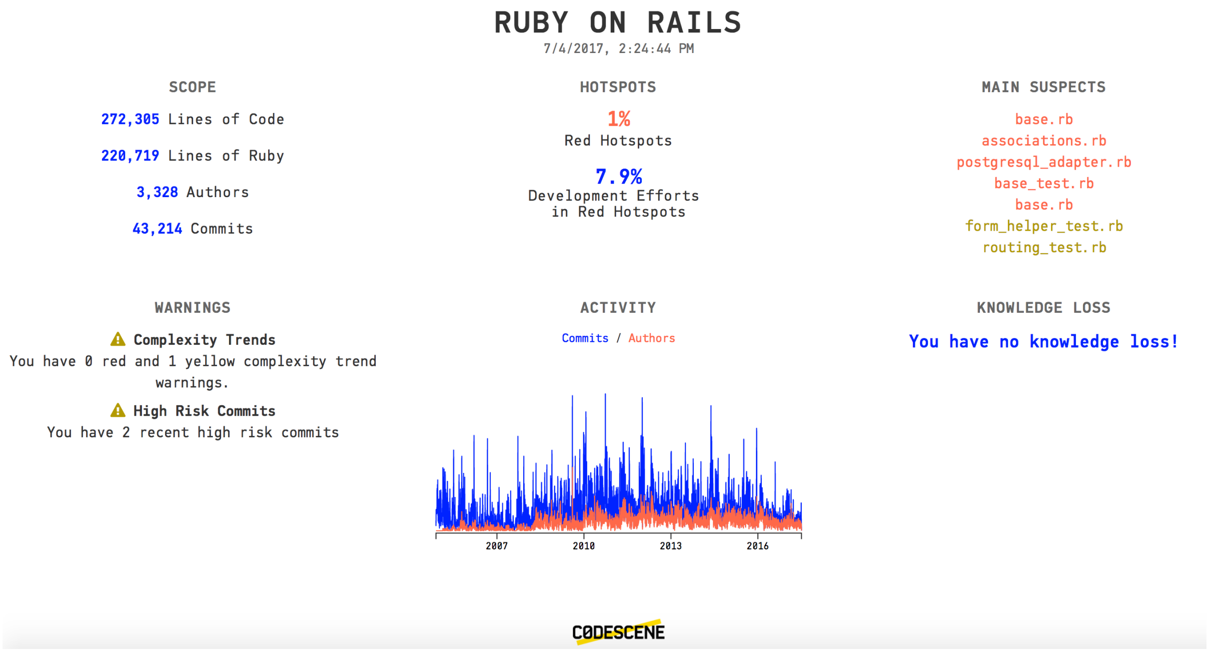
CodeScene presents an additional monitor view that is continuously updated with the status of your ongoing work on different feature branches. Present it on a TV in the office and use the information to drive code inspections and highlight potential delivery risks, as shown in Fig. 1.13.

The branch monitor displays all branches that haven't been merged yet.

Launch the branch monitor form the history view as shown in Fig. 1.14.

1.7 Upgrade Your License

Note: To get a new license key or upgrade existing license limitations (number of active authors) it's best to use our Customer Portal:.



RUBY ON RAILS
7/4/2017, 2:24:44 PM

SCOPE
272,305 Lines of Code
220,719 Lines of Ruby
3,328 Authors
43,214 Commits

HOTSPOTS
1% Red Hotspots
7.9% Development Efforts in Red Hotspots

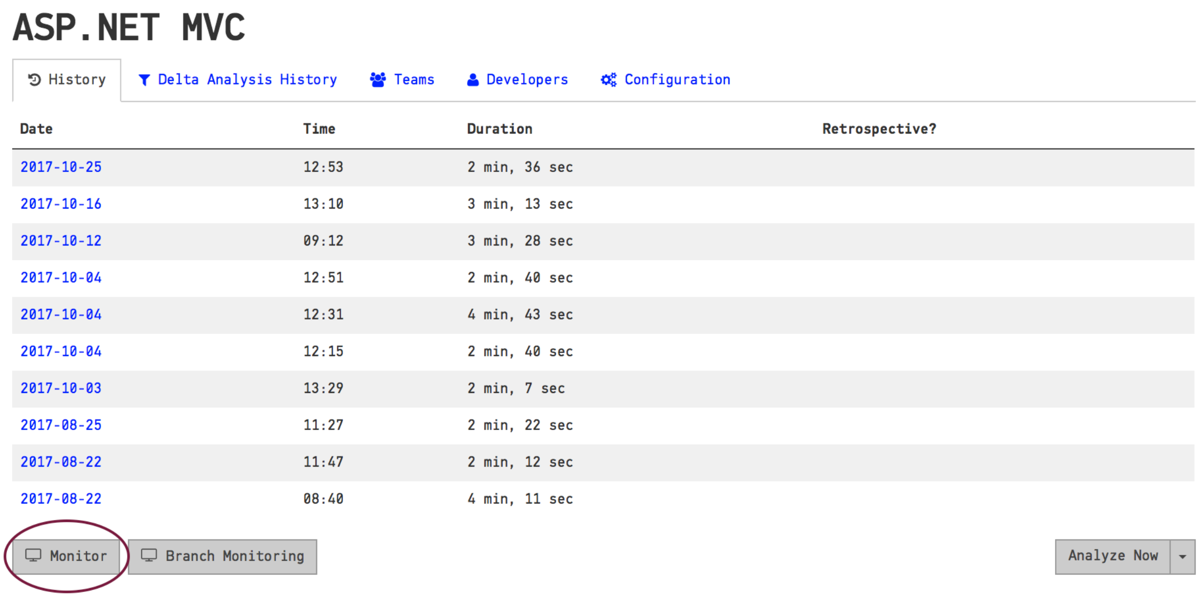
MAIN SUSPECTS
base.rb
associations.rb
postgresql_adapter.rb
base_test.rb
base.rb
form_helper_test.rb
routing_test.rb

WARNINGS
Complexity Trends
You have 0 red and 1 yellow complexity trend warnings.
High Risk Commits
You have 2 recent high risk commits

ACTIVITY

KNOWLEDGE LOSS
You have no knowledge loss!

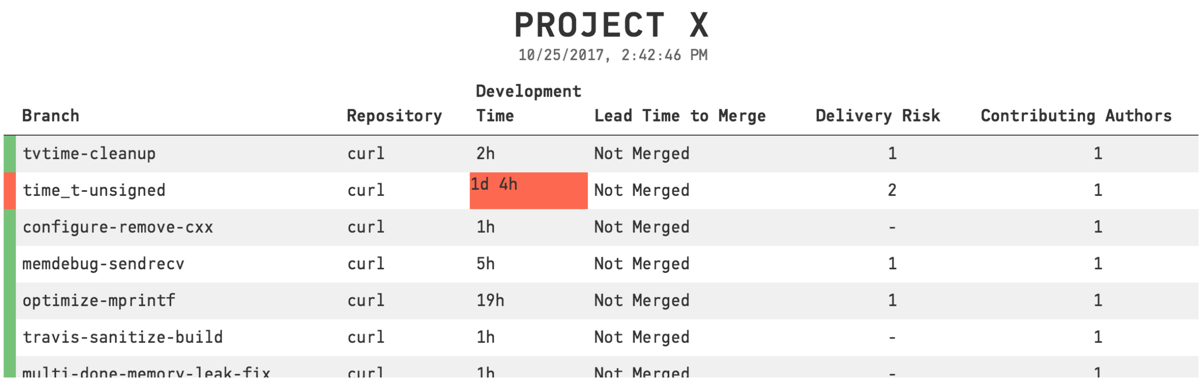
Fig. 1.11: The monitor dashboard gives you a high-level overview of your codebase.



ASP.NET MVC

Date	Time	Duration	Retrospective?
2017-10-25	12:53	2 min, 36 sec	
2017-10-16	13:10	3 min, 13 sec	
2017-10-12	09:12	3 min, 28 sec	
2017-10-04	12:51	2 min, 40 sec	
2017-10-04	12:31	4 min, 43 sec	
2017-10-04	12:15	2 min, 40 sec	
2017-10-03	13:29	2 min, 7 sec	
2017-08-25	11:27	2 min, 22 sec	
2017-08-22	11:47	2 min, 12 sec	
2017-08-22	08:40	4 min, 11 sec	

Fig. 1.12: Access the monitor dashboard from the History view in the project configuration.

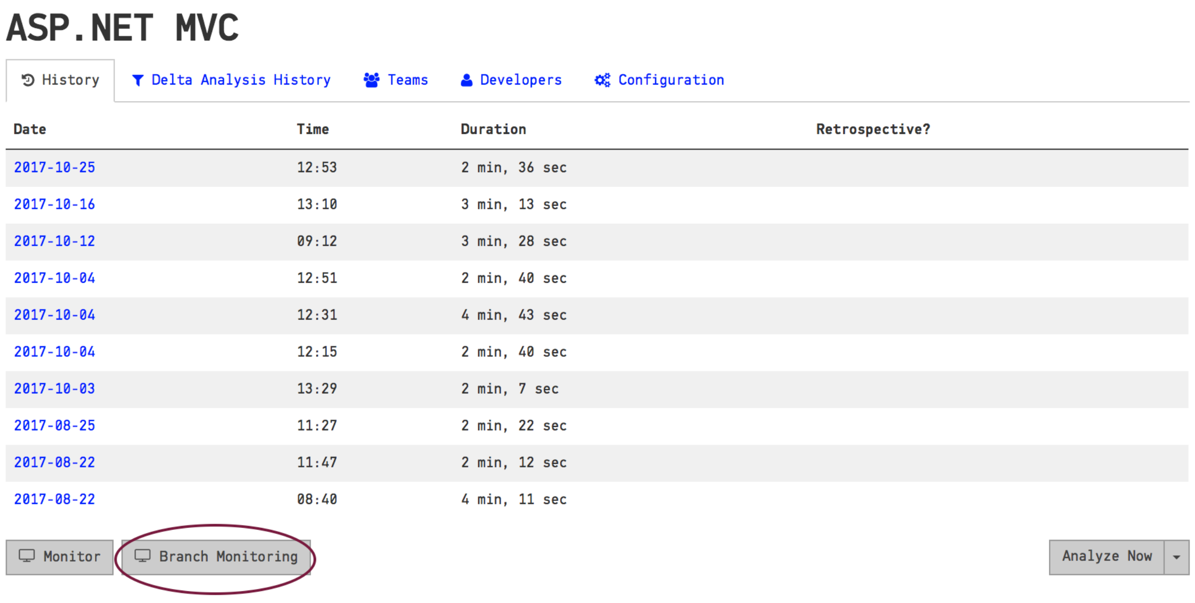


PROJECT X

10/25/2017, 2:42:46 PM

	Branch	Repository	Development Time	Lead Time to Merge	Delivery Risk	Contributing Authors
🟩	tvtime-cleanup	curl	2h	Not Merged	1	1
🟥	time_t-unsigned	curl	1d 4h	Not Merged	2	1
🟩	configure-remove-cxx	curl	1h	Not Merged	-	1
🟩	memdebug-sendrecv	curl	5h	Not Merged	1	1
🟩	optimize-mprintf	curl	19h	Not Merged	1	1
🟩	travis-sanitize-build	curl	1h	Not Merged	-	1
🟩	multi-done-memory-leak-fix	curl	1h	Not Merged	-	1

Fig. 1.13: Predict the delivery risk of each branch.



ASP.NET MVC

Date	Time	Duration	Retrospective?
2017-10-25	12:53	2 min, 36 sec	
2017-10-16	13:10	3 min, 13 sec	
2017-10-12	09:12	3 min, 28 sec	
2017-10-04	12:51	2 min, 40 sec	
2017-10-04	12:31	4 min, 43 sec	
2017-10-04	12:15	2 min, 40 sec	
2017-10-03	13:29	2 min, 7 sec	
2017-08-25	11:27	2 min, 22 sec	
2017-08-22	11:47	2 min, 12 sec	
2017-08-22	08:40	4 min, 11 sec	

Fig. 1.14: Launch the branch monitor

Following instructions show how to active the license key you receive in CodeScene. To upgrade from a trial license (or to a higher license category) you can simply and your license key will remain the same.

1.7.1 Upgrade from an Expired License

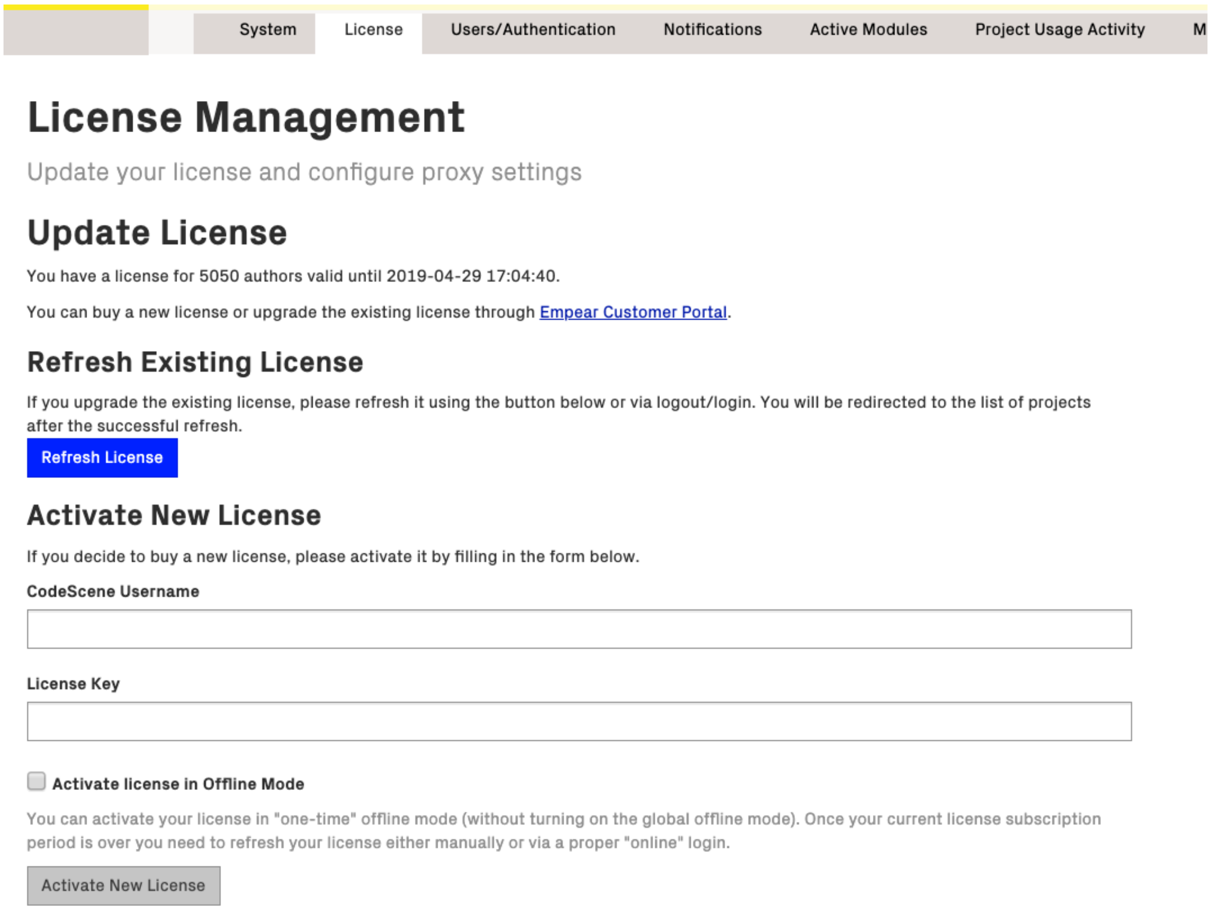
CodeScene will automatically prompt you for a new license once an existing license expires. Just enter your new credentials and everything will be up and running again. All your analyses and user configurations are preserved so you can login with any user after the license upgrade.

1.7.2 Upgrade from a Previous License

You may already have an activated instance of CodeScene running. To upgrade from a trial license (or to a higher license category) you can simply use our Customer Portal: and your license key will remain the same. Note that you'l need a working Internet connection to propagate changes from our license server to your CodeScene installation.

Alternatively (especially if you aren in *offline mode*), you can request a new license key: and activate the new license key as follows:

1. *Login as an administrator.* Login with the credentials from your *existing* license to get administration privileges.
2. *Click on 'Configuration' in the top menu* and go to the *License* tab as illustrated by Fig. 1.15.
3. Enter the new license credentials you received from Empear.
4. Press the *Activate New License* button and your new license becomes activated.



License Management

Update your license and configure proxy settings

Update License

You have a license for 5050 authors valid until 2019-04-29 17:04:40.

You can buy a new license or upgrade the existing license through Empear Customer Portal.

Refresh Existing License

If you upgrade the existing license, please refresh it using the button below or via logout/login. You will be redirected to the list of projects after the successful refresh.

Activate New License

If you decide to buy a new license, please activate it by filling in the form below.

CodeScene Username

License Key

Activate license in Offline Mode

You can activate your license in "one-time" offline mode (without turning on the global offline mode). Once your current license subscription period is over you need to refresh your license either manually or via a proper "online" login.

Fig. 1.15: Enter your new license credentials on the Configuration page.

Chapter 2

Hands On Behavioral Code Analysis: CodeScene Use Cases

CodeScene is different from the traditional code analysis tools you might have come across earlier. So follow along as we explain how you use the analysis information and how you integrate CodeScene in your organization's daily work to get the most out of the tool.

2.1 What is a Behavioral Code Analysis?

> Behavioral Code Analysis was pioneered by the research and techniques from the books Your Code As A Crime Scene and Software Design X-Rays: Fix Technical Debt with Behavioral Code Analysis by Adam Tornhill, Empear's founder.

The main difference between CodeScene's behavioral code analysis and traditional code scanning techniques is that static analysis works on a snapshot of the codebase while CodeScene considers the temporal dimension and evolution of the whole system. This makes it possible for CodeScene to prioritize technical debt and code quality issues based on how the organization actually works with the code (we'll give specific examples soon – promise).

In addition, CodeScene goes beyond code as it considers the organization and people side of the system. This gives you valuable information that is invisible in the source code itself, such as measures of team autonomy and off-boarding risks; factors that increase in importance with the scale of the development organization.

With the differences covered, let's see how to get started.

2.2 The Two Main Use Cases for Behavioral Code Analysis

CodeScene is used for two main use cases that are independent of each other:

1. *Optimize Your Software Delivery*: CodeScene comes with all the tools and features you need in order to optimize your software delivery. This includes analyses to prioritize technical debt, measure improvements, detect delivery risks and inter-team coordination bottlenecks, and much more. This information is used regularly and, typically, integrated in your organizations daily work.

2. *Ad-Hoc Strategic Analyses*: Some analyses in CodeScene aren't intended for daily use, but are there to support you when you need data to guide and support your decisions. These analyses include CodeScene's *Team Planning with the On- and Off-Boarding Simulation* (page 137) that lets you mitigate product risks, and CodeScene's change coupling analyses (see *Architectural Analyses* (page 88) that help you evaluate and optimize your software architecture based on how your system evolves.

In this tutorial we will focus on the first use case, and leave the second one to the *Guides* (page 36).

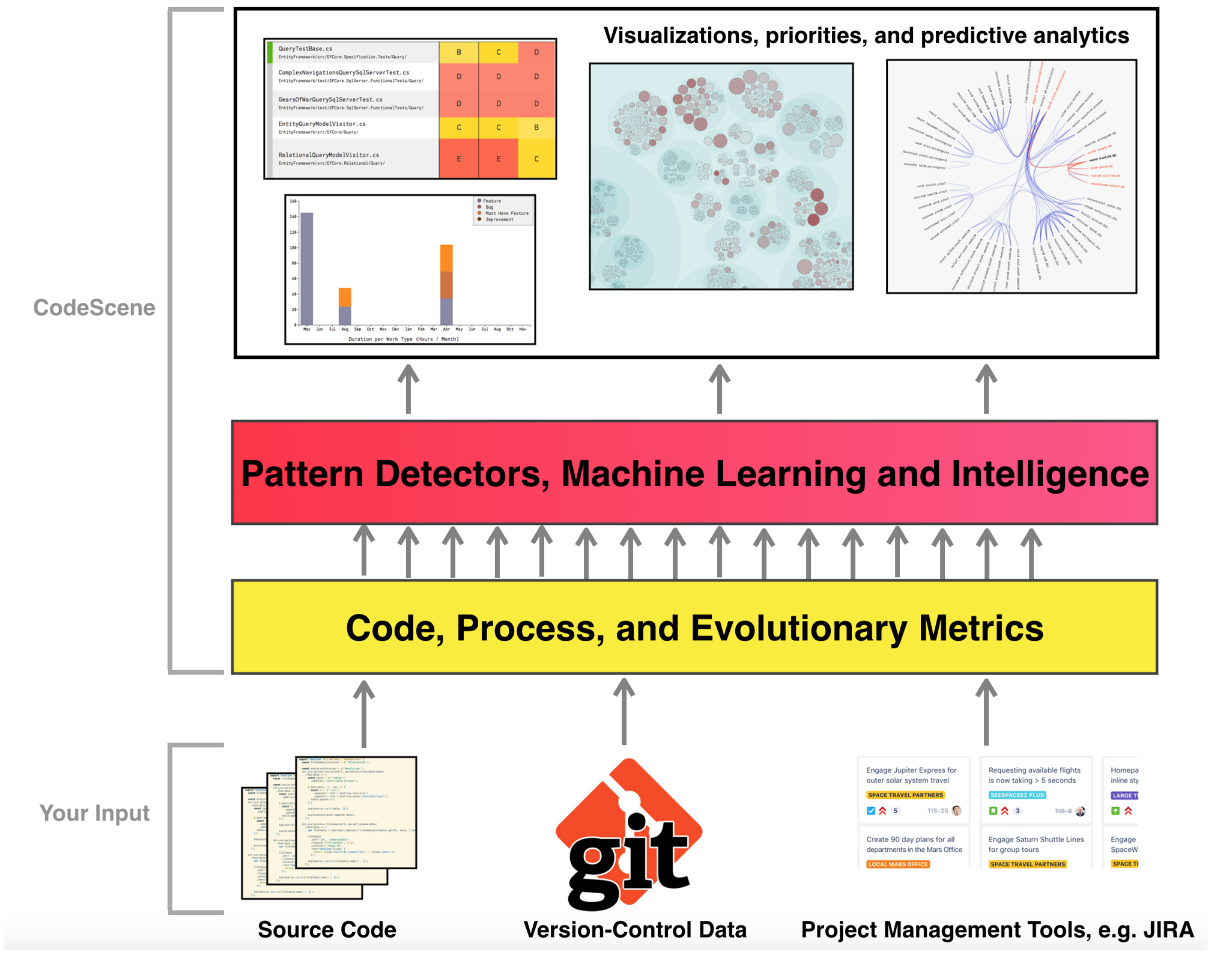

CodeScene
Visualizations, priorities, and predictive analytics
Pattern Detectors, Machine Learning and Intelligence
Code, Process, and Evolutionary Metrics
Your Input
Source Code
Version-Control Data
Project Management Tools, e.g. JIRA

Fig. 2.1: Behavioral Code Analysis builds on two main data sources plus an optional integration with project management tools like Jira.

2.3 A Workflow to Manage Technical Debt

Most of the time you will navigate via the main analysis dashboard in CodeScene, shown in Fig. 2.2. To learn more about the dashboards, check out *CodeScene's Dashboards: The Status of Your Codebase at a Glance* (page 36).

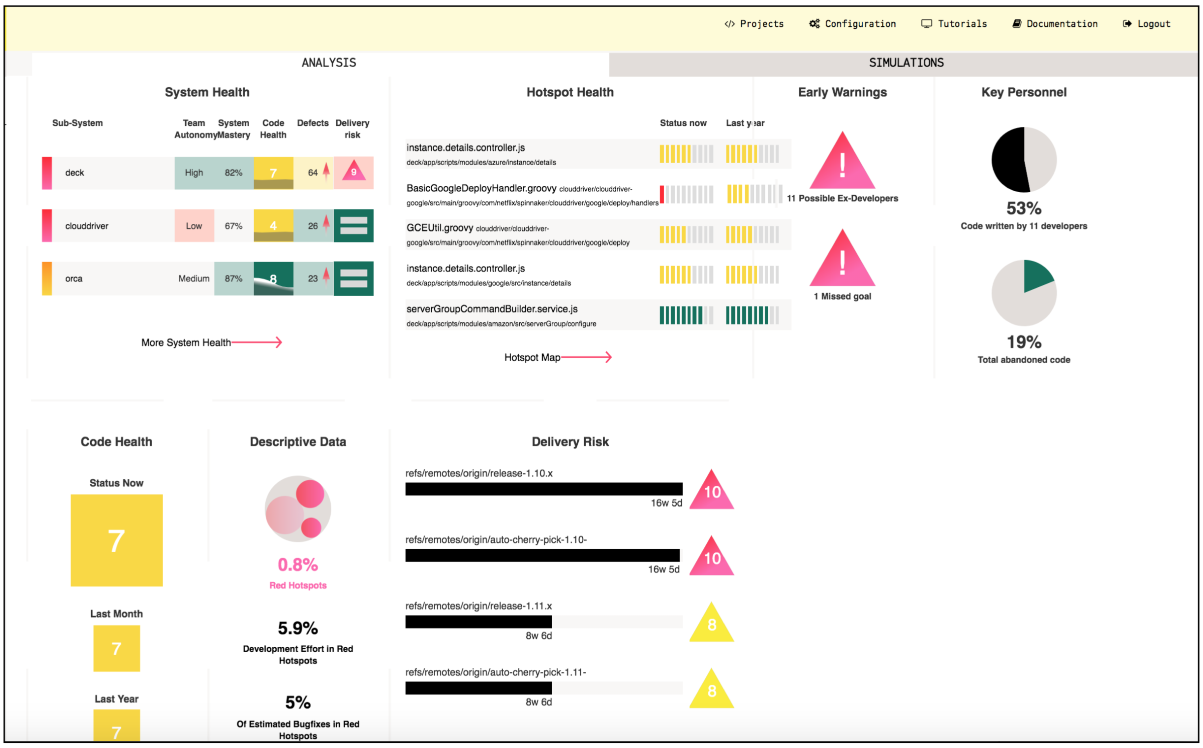

Projects
Configuration
Tutorials
Documentation
Logout
ANALYSIS
SIMULATIONS
System Health
Sub-System
Team Autonomy
System Mastery
Code Health
Defects
Delivery risk
deck
High
82%
7
64
9
clouddriver
Low
67%
4
26
orca
Medium
87%
8
23
More System Health
Hotspot Health
Status now
Last year
instance.details.controller.js
deck/app/scripts/modules/azure/instance/details
BasicGoogleDeployHandler.groovy clouddriver/clouddriver-google/src/main/groovy/com/netflix/spinnaker/clouddriver/google/deploy/handlers
GCEUtil.groovy clouddriver/clouddriver-google/src/main/groovy/com/netflix/spinnaker/clouddriver/google/deploy
instance.details.controller.js
deck/app/scripts/modules/google/src/instance/details
serverGroupCommandBuilder.service.js
deck/app/scripts/modules/amazon/src/serverGroup/configure
Hotspot Map
Early Warnings
11 Possible Ex-Developers
1 Missed goal
Key Personnel
53%
Code written by 11 developers
19%
Total abandoned code
Code Health
Status Now
7
Last Month
7
Last Year
7
Descriptive Data
0.8%
Red Hotspots
5.9%
Development Effort in Red Hotspots
5%
Of Estimated Bugfixes in Red Hotspots
Delivery Risk
refs/remotes/origin/release-1.10.x
16w 5d
10
refs/remotes/origin/auto-cherry-pick-1.10-
16w 5d
10
refs/remotes/origin/release-1.11.x
8w 6d
8
refs/remotes/origin/auto-cherry-pick-1.11-
8w 6d
8

Fig. 2.2: CodeScene's interactive analysis dashboard presents the key metrics at a glance.

However, you will use CodeScene differently the first times you run an analysis and start to explore a new codebase. Generally, a behavioral code analysis passes through three different stages:

1. The first time you analyse a codebase you spend some time building up a holistic overview of the code. As part of this stage, you use CodeScene to identify and prioritize technical debt. This stage is partly exploratory but CodeScene guides you along the way.

2. In the second stage your organization acts on the findings you have identified. You use CodeScene to follow-up and measure the effect of any code improvements or organizational changes to ensure you get a real – measureable – effect.

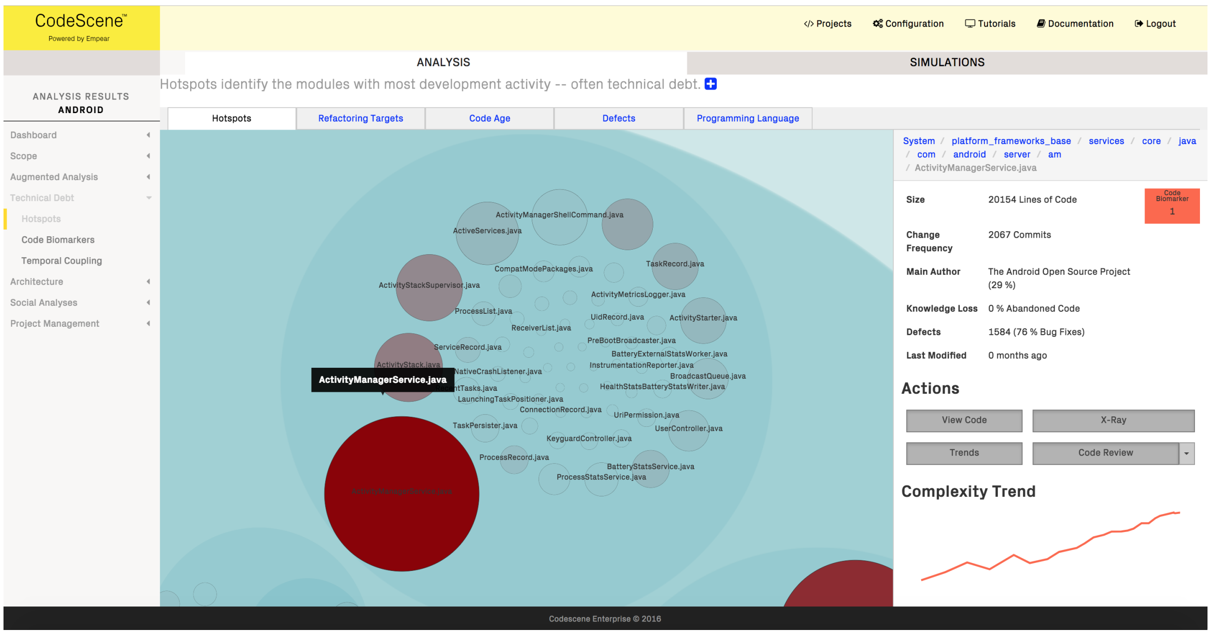
3. In the third stage, your technical debt is known, measured, and you use CodeScene to supervise it to ensure that no unexpected reckless debt is taken on. You might also use CodeScene to optimize your delivery by prioritizing code to review or direct testing activities based on CodeScene's risk prediction algorithms.

All of these stages are reflected in CodeScene. Let's start with the first one, technical debt prioritization.

2.3.1 Identify and Prioritize Technical Debt in Your First Analyses

We recommend starting by exploring your hotspots using the interactive map (see *Hotspots* (page 37)), as shown in Fig. 2.3.

The interactive map helps you build a high-level mental model of what the code looks like. From there, you can zoom in on the hotspots. The *Code Health* score helps you make a quick assessment on the quality of the hotspot code. In general, if a hotspot's Code Health is below *8*, it's worth digging deeper.



Hotspots identify the modules with most development activity -- often technical debt.

System / platform_frameworks_base / services / core / java / com / android / server / am / ActivityManagerService.java

Code Biomarker 1

Size	20154 Lines of Code
Change Frequency	2067 Commits
Main Author	The Android Open Source Project (29 %)
Knowledge Loss	0 % Abandoned Code
Defects	1584 (76 % Bug Fixes)
Last Modified	0 months ago

Actions

Complexity Trend

Fig. 2.3: Explore your codebase through the interactive hotspot map.

The reason you want to focus on hotspots is because they identify the parts of your code where most of your development activity is as measured by version-control data. You see an example in Fig. 2.4.

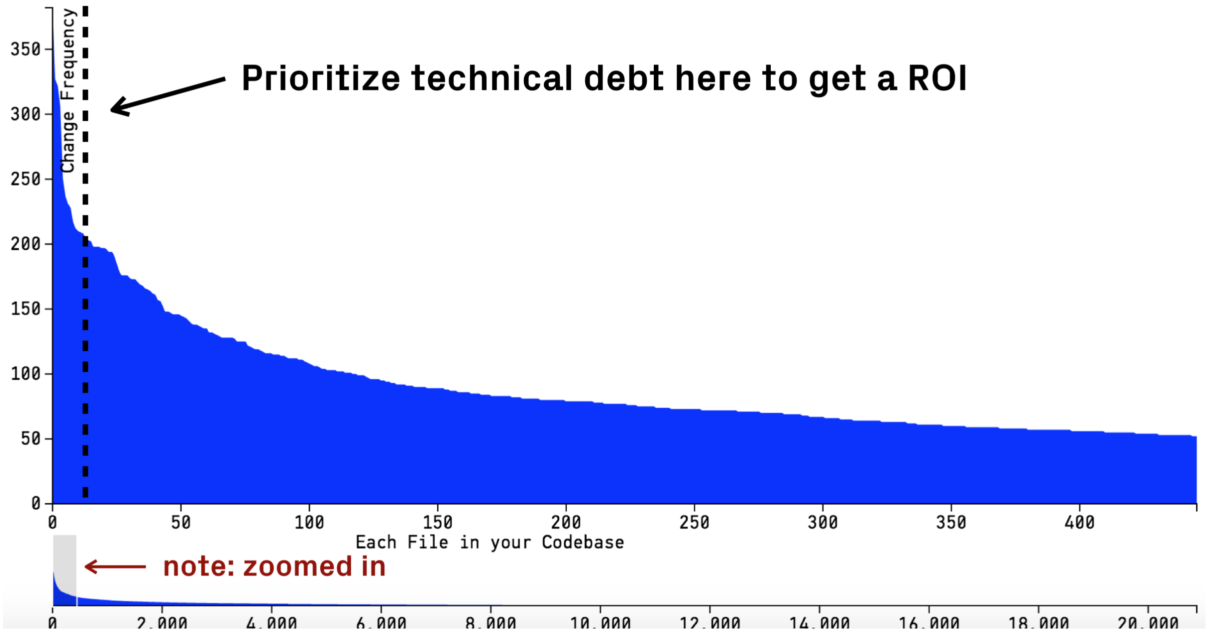


Fig. 2.4: All codebases evolve according to a power law, which means most development activity is in a relatively small part of the code.

Fig. 2.4 shows the change frequency of each module in a codebase. As you see, the change frequency forms a *power law* curve, and this is a pattern we at Empear have found in every single codebase we have analyzed; it is the nature of software evolution. What this means to you, is that most development activity is going to be in a relatively small part of your codebase. Hotspots identify those parts.

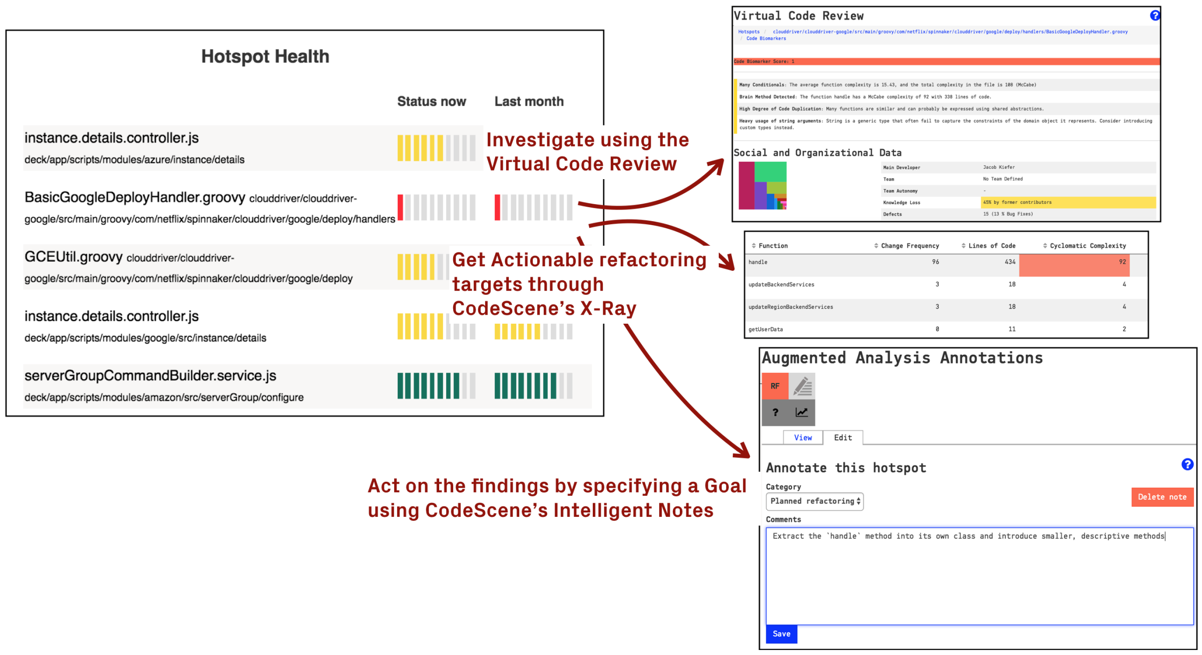
> A hotspot is the code where most of your teams development activity is. Hence, any technical debt you might identify there is likely to have a high interest rate. Paying off the technical debt in a hotspot gives you the most return on code quality improvements.

Once you have identified your top hotspots, you act upon the findings. Let's explore how CodeScene supports you in this stage.

2.3.2 Act on the Identified Technical Debt

Now, here's an important aspect of hotspots: just because a piece of code is worked on frequently, that doesn't mean it's a problem. If you inspect your top hotspots and the code in them looks good, then you are in a very good position; the code you work with the most is healthy, which means it's easy to evolve and understand. However, more often than not, the opposite tends to be true.

To find out if there is any technical debt in your hotspots, you request a *Virtual Code Review* (see *Code Biomarkers–A Virtual Code Reviewer aware of Code Health* (page 48)). You then run an X-Ray analysis to get specific, actionable findings in case the identified hotspot is a large file (see *X-Ray* (page 71)). Fig. 2.5 shows one example.


Hotspot Health
Status now
Last month
instance.details.controller.js
deck/app/scripts/modules/azure/instance/details
BasicGoogleDeployHandler.groovy clouddriver/clouddriver-google/src/main/groovy/com/netflix/spinnaker/clouddriver/google/deploy/handlers
GCEUtil.groovy clouddriver/clouddriver-google/src/main/groovy/com/netflix/spinnaker/clouddriver/google/deploy
instance.details.controller.js
deck/app/scripts/modules/google/src/instance/details
serverGroupCommandBuilder.service.js
deck/app/scripts/modules/amazon/src/serverGroup/configure
Investigate using the Virtual Code Review
Get Actionable refactoring targets through CodeScene's X-Ray
Act on the findings by specifying a Goal using CodeScene's Intelligent Notes
Virtual Code Review
Hotspots / clouddriver/clouddriver-google/src/main/groovy/com/netflix/spinnaker/clouddriver/google/deploy/handlers/BasicGoogleDeployHandler.groovy / Code Biomarkers
Social and Organizational Data
Augmented Analysis Annotations
RF
?
View
Edit
Annotate this hotspot
Delete note
Save

Code Biomarker Score: 1

Many Conditionals: The average function complexity is 15.43, and the total complexity in the file is 108 (McCabe)

Brain Method Detected: The function handle has a McCabe complexity of 92 with 338 lines of code.

High Degree of Code Duplication: Many functions are similar and can probably be expressed using shared abstractions.

Heavy usage of string arguments: String is a generic type that often fail to capture the constraints of the domain object it represents. Consider introducing custom types instead.

Main Developer	Jacob Kiefer
Team	No Team Defined
Team Autonomy	-
Knowledge Loss	45% by former contributors
Defects	15 (13 % Bug Fixes)

Function	Change Frequency	Lines of Code	Cyclomatic Complexity
handle	96	434	92
updateBackendServices	3	18	4
updateRegionBackendServices	3	18	4
getUserData	0	11	2

Category: Planned refactoring

Comments: Extract the `handle` method into its own class and introduce smaller, descriptive methods

Fig. 2.5: Act on the hotspots by using the deeper analyses, and capture the outcome of your investigation in a goal.

Using CodeScene's *Code Health* score, you get a quick assessment on how healthy the code is. A low health, e.g. anything below ‘7, is likely to be expensive in terms of future development as any technical debt in a hotspot comes with a high interest rate.

The key is to use CodeScene to dive deeper into a hotspot, and then capture the outcome of your investigation in a goal. Goals are specified using CodeScene's *Intelligent Notes*, which is a core concept that makes technical debt both transparent, contextual, and – most important – actionable.

> Specify Goals for each hotspot using CodeScene's Intelligent Notes – the Goals enable several other important features.

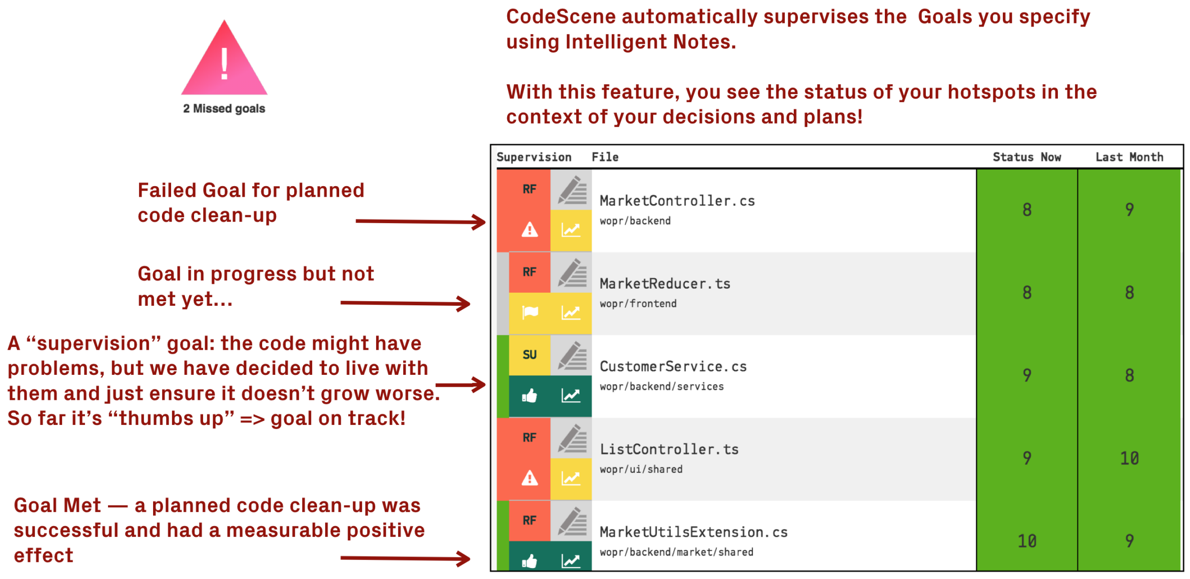
Intelligent Notes are described in detail in *Augmented Analysis with Intelligent Notes* (page 54). Use goals to classify each hotspot as either:

1. code that needs to be cleaned-up (*Plan Refactoring*),
2. code that might have issues, but is good enough for now. We just want to ensure it doesn't get worse (*Supervise*), or
3. code that is OK as-is, so you no longer wish to see it as a hotspot *unless* something dramatic happens to the code (*No Problem*).

Whatever goal you specify, CodeScene will supervise those goals and inform you on their progress. Let's see some examples.

2.3.3 Track and Visualize the State of your Technical Debt

As soon as you specify one or more goals, CodeScene will start to measure against them. You will see a high-level summary on the dashboards, and also get specific reports on each hotspot as shown in Fig. 2.6



CodeScene automatically supervises the Goals you specify using Intelligent Notes.

With this feature, you see the status of your hotspots in the context of your decisions and plans!

Failed Goal for planned code clean-up

Goal in progress but not met yet...

A "supervision" goal: the code might have problems, but we have decided to live with them and just ensure it doesn't grow worse. So far it's "thumbs up" => goal on track!

Goal Met — a planned code clean-up was successful and had a measurable positive effect

Supervision	File	Status Now	Last Month
RF	MarketController.cs wopr/backend	8	9
RF	MarketReducer.ts wopr/frontend	8	8
SU	CustomerService.cs wopr/backend/services	9	8
RF	ListController.ts wopr/ui/shared	9	10
RF	MarketUtilsExtension.cs wopr/backend/market/shared	10	9

Fig. 2.6: Goals let you see the status of your hotspots in the context of your decisions and plans.

With the assistance of Intelligent Notes, you will see the status of your hotspots in the context of your decisions and plans. CodeScene will also present a warning on the analysis dashboard in case a goal is violated. However, feedback loops should be short so CodeScene offers additional warnings. Let's look briefly at how you integrate this in a Continuous Integration/Delivery pipeline.

2.3.4 Supervise Your Goals with CI/CD Quality Gates

The earlier you can react to any potential problem or surprise, the better. That's why CodeScene offers integration points that let you incorporate the analysis results into your build pipeline (see *CI/CD Integration with CodeScene's Delta Analysis* (page 123)).

> Integrate CodeScene in your build pipeline to supervise your goals and prevent that unplanned technical debt is taken on.

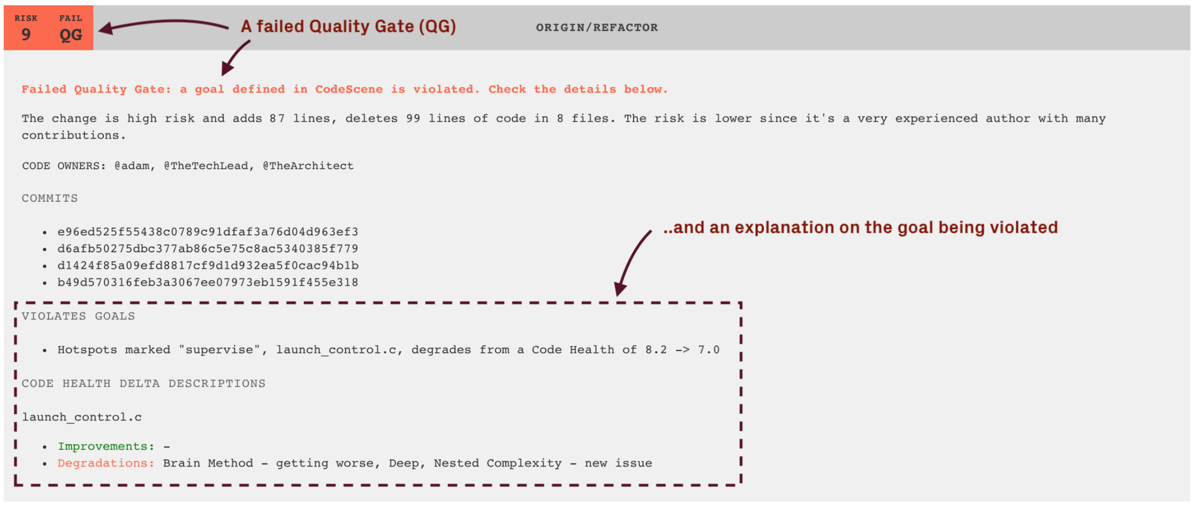
The CI/CD integration triggers a special type of analysis called a *delta analysis*. A delta analysis is quick. The main purpose of this integration is to:

1. Predict the delivery risk of a specific change set in order to prioritize code reviews and testing activites.
2. Supervise your goals using Quality Gates to ensure that none of the goals you have specified via Intelligent Notes are violated.

That way, violated goals can be detected before they become an issue. In addition, the status of your goals is also presented on CodeScene's inter-product dashboard. Let's look at an example.

2.4 Track Multiple Codebases and Products on the Inter-Project Dashboard

CodeScene also presents an inter-project dashboard that gives you a high-level overview of all your products, as shown in Fig. 2.8.



RISK 9 FAIL QG

A failed Quality Gate (QG)

ORIGIN/REFACTOR

Failed Quality Gate: a goal defined in CodeScene is violated. Check the details below.

The change is high risk and adds 87 lines, deletes 99 lines of code in 8 files. The risk is lower since it's a very experienced author with many contributions.

CODE OWNERS: @adam, @TheTechLead, @TheArchitect

COMMITS

- e96ed525f55438c0789c91dfaf3a76d04d963ef3
- d6afb50275dbc377ab86c5e75c8ac5340385f779
- d1424f85a09efd8817cf9d1d932ea5f0cac94b1b
- b49d570316feb3a3067ee07973eb1591f455e318

..and an explanation on the goal being violated

VIOLATES GOALS

- Hotspots marked "supervise", launch_control.c, degrades from a Code Health of 8.2 -> 7.0

CODE HEALTH DELTA DESCRIPTIONS

launch_control.c

- Improvements: -
- Degradations: Brain Method - getting worse, Deep, Nested Complexity - new issue

Fig. 2.7: Your goals are supervised in the CI/CD pipeline.

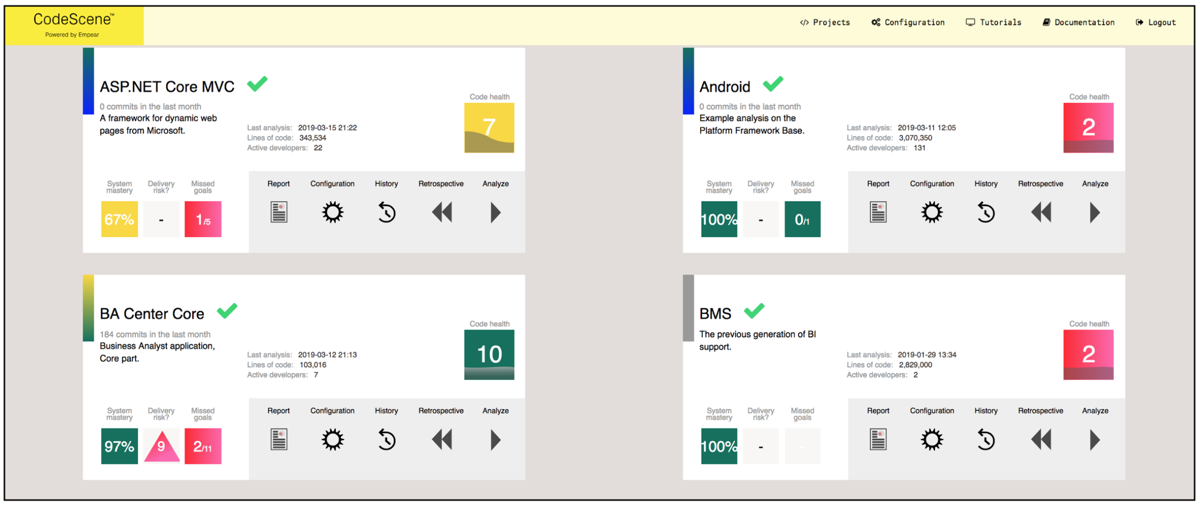


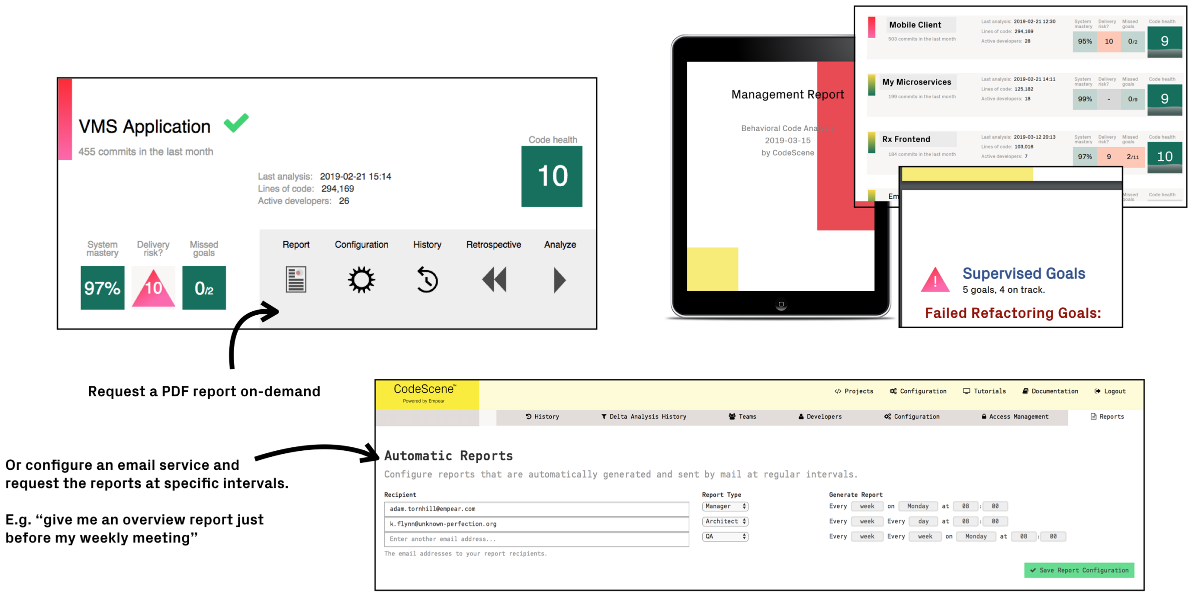
Fig. 2.8: The high-level dashboard shows that status of all your products at a glance.

Using these key metrics on the dashboard, in particular the progress on the goals, point you the products/codebase that you need to inspect in more depth.

You can also restrict access to certain analysis projects. Checkout *CodeScene's Dashboards: The Status of Your Codebase at a Glance* (page 36) for more details and use cases.

2.5 Subscribe to Auto-Generated Analysis Reports

CodeScene also offers auto-generated PDF reports adapted to different roles such as managers, architects, and lead developers. Using those reports, each stakeholder gets the most important information delivered without having to sign into CodeScene.



Automatic Reports

Configure reports that are automatically generated and sent by mail at regular intervals.

Recipient	Report Type	Generate Report
adam.tornhill@empear.com	Manager	Every week on Monday at 08 : 00
k.flynn@unknown-perfection.org	Architect	Every week Every day at 08 : 00
Enter another email address...	QA	Every week Every week on Monday at 08 : 00

The email addresses to your report recipients.

Fig. 2.9: *Get an auto-generated analysis report emailed to you as a PDF or create it on-demand.*

The reports too will highlight the progress of your planned goals and intelligent notes. You can create the on-demand from the inter-product dashboard, or configure CodeScene to automatically email them to you at specific intervals.

2.6 What's Next?

This introduction has covered the main usage of CodeScene, which is to prioritize and act on technical debt. By following the recommended steps in this introduction, you will soon have your technical debt and the future of your codebase firmly under control.

To dive deeper, we recommend that you also check-out the following analyses:

- *Temporal Coupling* (page 60)
- The Conway's Law Analysis and the *Architectural Analyses* (page 88)
- *Team Planning with the On- and Off-Boarding Simulation* (page 137)

Chapter 3

Integrations

CodeScene comes with a set of integrations that let you integrate the analysis information into your daily workflows. Use the CI/CD integration for quality gates and to streamline code reviews. Use the Jira integration to measure Lead Times and Throughput.

3.1 Integrate CodeScene in your CI/CD Pipeline

By integration CodeScene into your CI/CD Pipeline and/or your code review tools, you get the following advantages:

- Prioritize code reviews based on the risk of the commits.
- Specify quality gates for the goals specified on identified hotspots using CodeScene's Intelligent Notes.
- Specify quality gates that trigger in case the Code Health of a hotspot declines.
- Get early warnings such a complexity trend increases and detect the absence of expected change coupling.

These use cases are explained in *CI/CD Integration with CodeScene's Delta Analysis* (page 123).

The rest of this guide explains the integrations for the supported platforms.

3.1.1 CodeScene Jenkins Plugin

CodeScene provides an official Jenkins plugin, available via Jenkins Update Center.

More detailed installation and configuration instructions are available here.

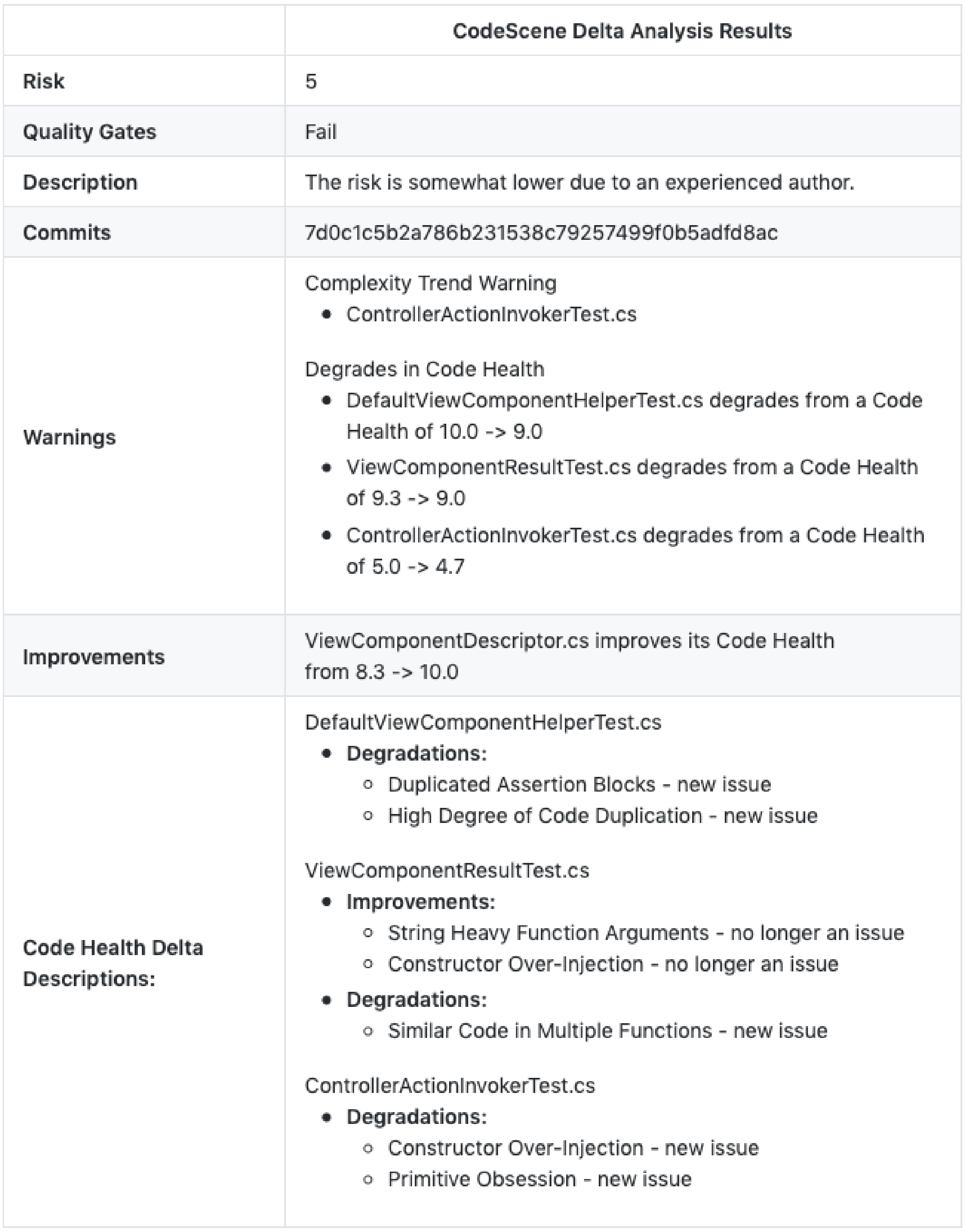
3.1.2 CodeScene's Automated Pull Request Review for GitHub, GitLab, and BitBucket

CodeScene integrates in a code review workflow to provide automatic review comments on pull requests. With this integration, any code health decline or goal violations are caught early.

To enable this integration, download the *codescene-ci-cd* bridge – with documentation – from here.

3.1.3 CodeScene Orb for CircleCI Integration

CodeScene provides an official CircleCI Orb, available directly via the CircleCI Orb Registry.



	CodeScene Delta Analysis Results
Risk	5
Quality Gates	Fail
Description	The risk is somewhat lower due to an experienced author.
Commits	7d0c1c5b2a786b231538c79257499f0b5adfd8ac
Warnings	Complexity Trend Warning • ControllerActionInvokerTest.cs Degrades in Code Health • DefaultViewComponentHelperTest.cs degrades from a Code Health of 10.0 -> 9.0 • ViewComponentResultTest.cs degrades from a Code Health of 9.3 -> 9.0 • ControllerActionInvokerTest.cs degrades from a Code Health of 5.0 -> 4.7
Improvements	ViewComponentDescriptor.cs improves its Code Health from 8.3 -> 10.0
Code Health Delta Descriptions:	DefaultViewComponentHelperTest.cs • **Degradations:** ◦ Duplicated Assertion Blocks - new issue ◦ High Degree of Code Duplication - new issue ViewComponentResultTest.cs • **Improvements:** ◦ String Heavy Function Arguments - no longer an issue ◦ Constructor Over-Injection - no longer an issue • **Degradations:** ◦ Similar Code in Multiple Functions - new issue ControllerActionInvokerTest.cs • **Degradations:** ◦ Constructor Over-Injection - new issue ◦ Primitive Obsession - new issue

Fig. 3.1: CodeScene provides automated pull request comments.

3.1.4 CodeScene Integration with Gerrit

You can integrate with Gerrit either by using CodeScene's Jenkins plugin as a code review bot (*+1*), or by querying CodeScene from Gerrit on demand. Here's how it works:

Gerrit provides a staging area for code to be reviewed. This staging area is kept separate from the main, authorative Git repository. As a consequence, the commits for a delta analysis aren't available in the main Git repository, but in Gerrit's mirror of the repository.

CodeScene lets you resolve this by specifying a different *origin_url* and a specific *change_ref* to fetch before the delta analysis is run. Here's an example:

```
curl -X POST -d '{"commits": ["149f9e6"], "repository": "PhpSpreadsheet", "origin_url":
↪"gerrit.mycompany.com:39429/dev/wopr", "change_ref": "refs/changes/82/577659/7"}' http://
↪localhost:3003/projects/64/delta-analysis -u 'CodeReview:MyPassword' -H "content-type:␣
↪application/json"
```

That is, CodeScene will fetch a specific change set from Gerrit and then run the delta analysis as indicated by the other parameters you provide.

3.2 Integrate Jira Information into CodeScene

CodeScene's Jira integration is optional, but highly recommended if you have the information required for the analyses:

- The Jira issue numbers are included/referenced in the commit messages.
- You use labels and/or issue types in Jira to distinguish different kinds of work (e.g. Bugs, Features).

When present, CodeScene's Jira integration lets you measure:

- Accumulated costs per hotspot and sub-system.
- Trends by work type, such as "Planned" versus "Unplanned" work.

  CodeScene's Jira integration lets you reason about the technical and organizational findings from a financial perspective. For example, how much time do you spend on defects in your top hotspots? What happens over time?

CodeScene supports multiple cost models depending on the data you have available such as cycle times, story points, or time spent. CodeScene can deduce costs automatically based on your development history, so you don't even need to have developers reporting their time to be able to get a detailed view of your development costs. We cover all options and provide our recommended setup in this guide. Let's start by looking at the overall model.

3.2.1 CodeScene's Cost Model

CodeScene offers a breakdown of the development costs on three separate levels: file-, architecture-, and system-level. The file level corresponds to the hotspots, the architecture level accumulates costs on a component/service/module level, whereas the system level presents the cost trend as aggregated for all application code.

CodeScene distributes the derived costs proportional to the change impact. That is, the modules with most work in a specific issue get proportionally higher costs than modules that had less work. CodeScene automatically deduces this distribution.

The costs for multiple issues are then aggregated and split according to the type of work.

Using CodeScene's cost model, you get the specific development costs on each hotspot. Use this data to inform re-work decisions and to communicate the costs of technical debt to the business:

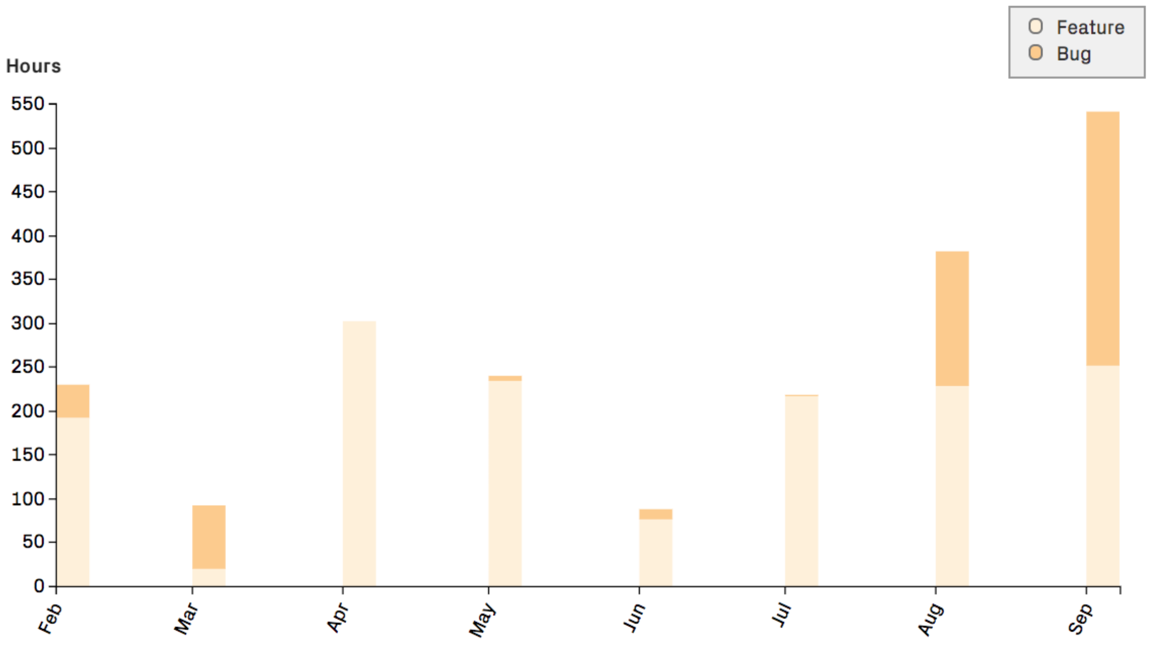
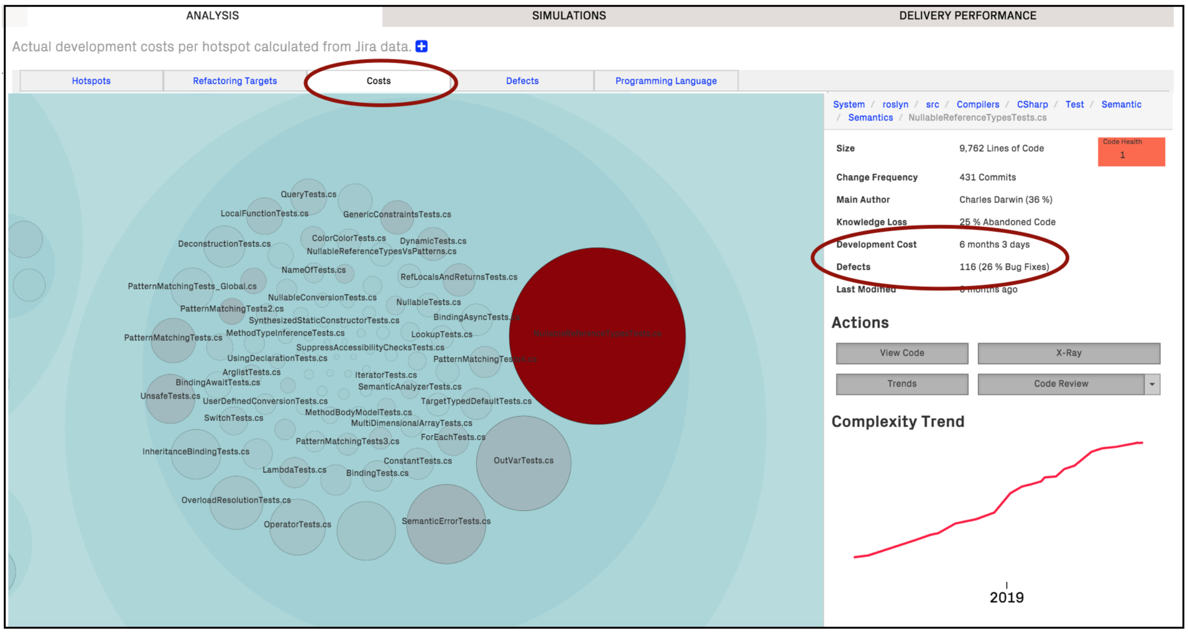


Fig. 3.2: CodeScene calculates cost trends on a file-, architecture-, and system-level.



Actual development costs per hotspot calculated from Jira data.

System / roslyn / src / Compilers / CSharp / Test / Semantic / Semantics / NullableReferenceTypesTests.cs

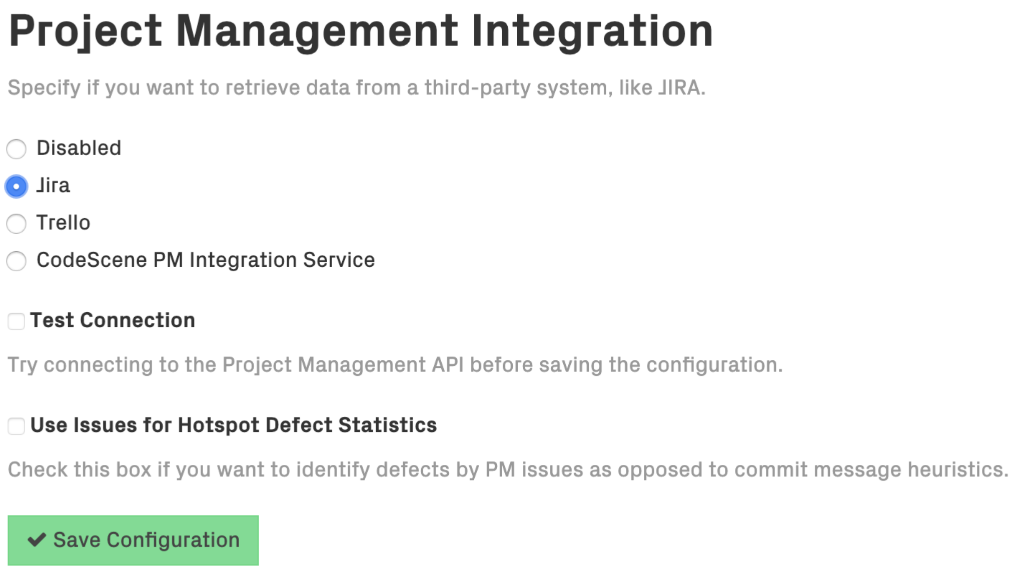
Size	9,762 Lines of Code
Change Frequency	431 Commits
Main Author	Charles Darwin (36 %)
Knowledge Loss	25 % Abandoned Code
Development Cost	6 months 3 days
Defects	116 (26 % Bug Fixes)
Last Modified	6 months ago

Fig. 3.3: CodeScene calculates cost per hotspot, including both feature and defect data.

3.2.2 Connect CodeScene to Jira

Previously, a separate service was required to connect CodeScene to Jira. Thi is no longer necessary because CodeScene can now connect directly to the Jira API. If you are still using the CodeScene Jira plugin, the documentation is available here: /integrations/deprecated-jira-service. However, we recommend using the new direct connection which also provides access to the latest analysis features.

Jira configuration is per project. Go to the "Project Management Integration" tab in your project's configuration. Select "Jira" and click on "Save Configuration":


Project Management Integration

Specify if you want to retrieve data from a third-party system, like JIRA.

( ) Disabled
[✓ Jira]
( ) Trello
( ) CodeScene PM Integration Service

☐ Test Connection

Try connecting to the Project Management API before saving the configuration.

☐ Use Issues for Hotspot Defect Statistics

Check this box if you want to identify defects by PM issues as opposed to commit message heuristics.

Fig. 3.4: Start by selecting "Jira"

When you save the configuation, the form for configuring your Jira connection will appear:

Fill in your Jira credentials here. We recommend using a Jira API token as the password. *External Project ID* is the Jira identifier for your project.

If an issue in your project looks like this, the *External Project ID* would be *FJ*.

Submitting the form with the "Test connection" box checked will cause CodeScene to immediately attempt to connect to your Jira provider.

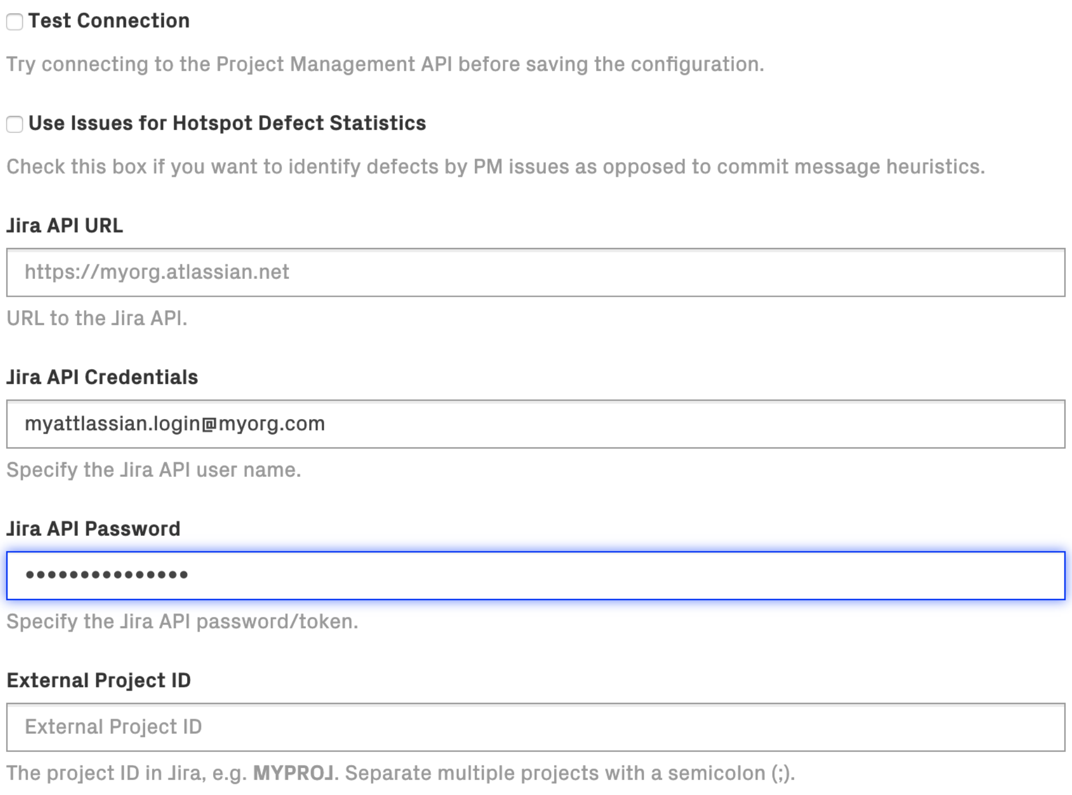
When *Use Issues for Hotspot Defect Statistics* is checked, CodeScene will rely on Jira information for identifying defects rather than matching strings in commit messages. This affects how CodeScene compiles statistics related to the frequency of bugs in a codebase.

The remaining fields concern how CodeScene will interpret the Jira issues in your project.

Specify a Cost Model

Cost Field and *Cost Unit* tell CodeScene how to calculate costs. There are four strategies which correspond to the possibilities in the *Cost Unit* field:

- *Cycle Time in Development*: the number of hours from the time an issue entered a specific Jira state until the last commit is done on this issue. The cycle time is adjusted for the number of hours in a typical work day. That is, a cycle time of 3 days has a cost of *3 * 8* hours. The Cycle Time option is the recommended default model since it lets CodeScene calculate costs as opposed to rely on time reported in Jira, which is often incorrect, incomplete, or both.



☐ **Test Connection**

Try connecting to the Project Management API before saving the configuration.

☐ **Use Issues for Hotspot Defect Statistics**

Check this box if you want to identify defects by PM issues as opposed to commit message heuristics.

Jira API URL

https://myorg.atlassian.net

URL to the Jira API.

Jira API Credentials

myattlassian.login@myorg.com

Specify the Jira API user name.

Jira API Password

•••••••••••••••

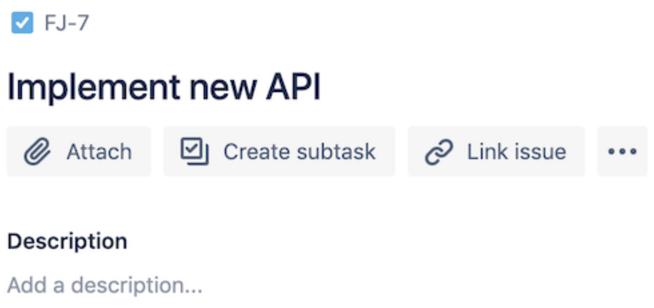
Specify the Jira API password/token.

External Project ID

External Project ID

The project ID in Jira, e.g. **MYPROJ**. Separate multiple projects with a semicolon (;).

Fig. 3.5: Jira configuration options



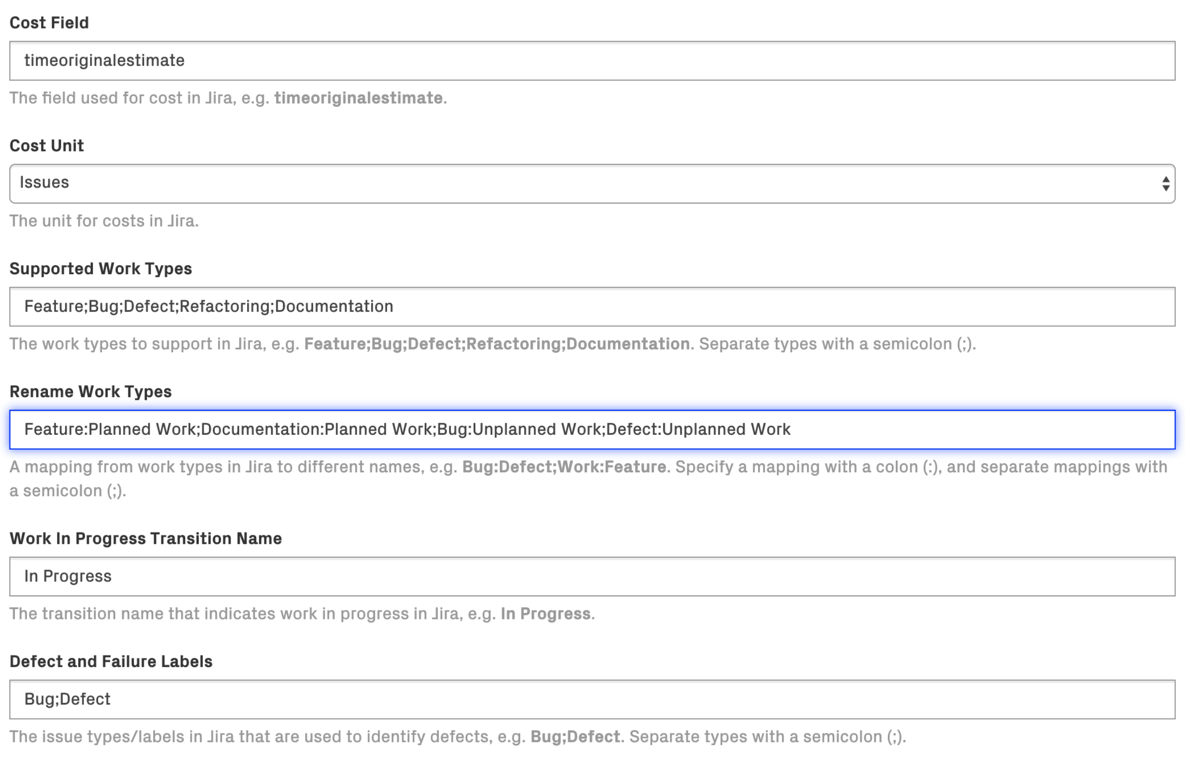
☑ FJ-7

Implement new API

Description

Add a description...

Fig. 3.6: "External Project ID" would be FJ



Cost Field

timeoriginalestimate

The field used for cost in Jira, e.g. **timeoriginalestimate**.

Cost Unit

Issues

The unit for costs in Jira.

Supported Work Types

Feature;Bug;Defect;Refactoring;Documentation

The work types to support in Jira, e.g. **Feature;Bug;Defect;Refactoring;Documentation**. Separate types with a semicolon (;).

Rename Work Types

Feature:Planned Work;Documentation:Planned Work;Bug:Unplanned Work;Defect:Unplanned Work

A mapping from work types in Jira to different names, e.g. **Bug:Defect;Work:Feature**. Specify a mapping with a colon (:), and separate mappings with a semicolon (;).

Work In Progress Transition Name

In Progress

The transition name that indicates work in progress in Jira, e.g. **In Progress**.

Defect and Failure Labels

Bug;Defect

The issue types/labels in Jira that are used to identify defects, e.g. **Bug;Defect**. Separate types with a semicolon (;).

Fig. 3.7: These fields determine how your data will be applied to the project

- *Issues*: the number of issues associated with a given file or architectural component. Use this to get a summary but without the detailed costs.
- *Points*: the cost of an Issue in /Story points/. Use this option if you keep track of story points and use them to communicate within the organization.
- *Minutes*: the cost expressed in time. This method requires that time has been reported on the specified cost-field in JIRA. Use this option if you have accurate data in JIRA on how much time you have spent on each issue.

To use *Issues* or *Cycle Time in Development*, it is not necessary to configure *Cost Field*. For *Points* or *Minutes*, see the sections *Using Story Points as Cost Estimation in Jira* and *Using Reported Time as Cost Estimation in Jira* below.

To use *Cycle Time in Development*, you need to configure a *Work In Progress Transition Name*. This should be the name of the Jira status that indicates that the development work has started. Often, this is the “In Progress” or “In Development” state.

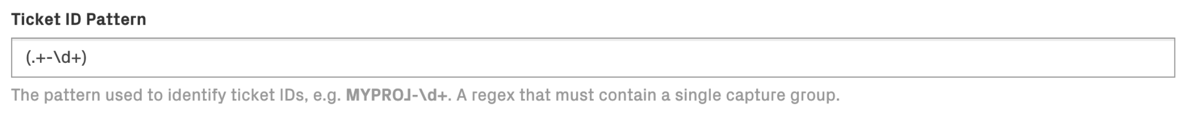
Configure the Types of Work

The *Supported Work Types* should correspond to the different kinds of issue labels defined in your Jira project, in a semi-colon separated list. Please note that only types with the listed labels/type will be included in the analysis.

The *Rename Work Types* field allows the work types to be mapped to different analytical categories that you can define yourself. How you do this depends on the type of analysis you wish to perform.

When looking at cost trends, the most interesting distinction is typically between Planned- versus Unplanned Work. When available, the Jira labels will be translated to the specified label before being sent to CodeScene. For example, if your Jira project contains *Feature* and *Documentation* labels, like in the illustration above, these can be categorized together as *Planned Work*, while *Bug* and *Defect* are treated as *Unplanned Work*; the *Refactoring* label – which doesn’t have a translation – will be sent as is. Mapping labels this way can allow you to see more meaningful trends. You are free to map labels however you like depending on your analytical needs.

The *Defect and Failure Labels* pecify the JIRA labels and/or JIRA Issue Types that will be regarded as defects. Note that this is independent from the *Supported Work Types* configuration (eg. a label added here will be used even if it is not present in *Supported Work Types*)


Ticket ID Pattern

(.+-\d+)

The pattern used to identify ticket IDs, e.g. **MYPROJ-\d+**. A regex that must contain a single capture group.

The *Ticket ID Pattern* is a regular expression that tells CodeScene how to extract the Jira ID from commit messages. For most projects that use the standard *PROJ-123* format, the default pattern will work: *(.+-d+)*.

3.2.3 Using Story Points as Cost Estimation in Jira

When JIRA is configured to use Story Points as estimate for stories, epics, and possibly other issue types, the points will be added in a custom field JIRA creates for this purpose when Story Points is configured. The custom field will get a generated id. In order to be able to configure the PM integration service correctly, this custom field needs to be identified. The following command (against the JIRA service) using *curl* and *jq* will filter out the custom field for a project with the key *CSE2* (see Atlassian Answers):

```
$ curl -u 'jirauser:jirapwd' \
'https://jira.example.com/rest/api/latest/issue/createmeta?expand=projects.issuetypes.fields'\
|jq '.projects[]|select(.key=="CSE2")|.issuetypes[]|select(.name=="Story")|.fields|with_
↪entries(select(.value.name=="Story Points"))'
{
  "customfield_10006": {
    "required": false,
    "schema": {
      "type": "number",
      "custom": "com.atlassian.jira.plugin.system.customfieldtypes:float",
      "customId": 10006
    },
    "name": "Story Points",
    "hasDefaultValue": false,
    "operations": [
      "set"
    ]
  }
}
```

You can verify that this is in fact the field with the Story Points. Say that you already have a story *CSE2-257* with *Estimate: 4* filled in, then you can find the field name and verify the points with this command:

```
$ curl -u 'jirauser:jirapwd' \
https://jira.example.com/rest/api/latest/issue/CSE2-257
{
  ...
  "fields": {
    ...
    "timetracking": {},
    "customfield_10006": 4,
```

3.2.4 Using Reported Time as Cost Estimation in Jira

When using Minutes instead of Points, CodeScene searches Jira for the configured field name, usually *timeoriginalestimate*, having a non-empty value:

```
curl -u 'jirauser:jirapwd' \
'https://jira.example.com/rest/api/latest/search?jql=project=CSE2+and+timeoriginalestimate!
↪=NULL'
```

Custom fields, however, cannot be searched like regular fields. Unfortunately, it seems it's not possible to just use the complete field name, *customfield_10006*:

```
$ curl -u 'jirauser:jirapwd' \
'https://jira.example.com/rest/api/latest/search?jql=project=CSE2+and+customfield_10006!=NULL'
{
  "errorMessages": [
    "Field 'customfield_10006' does not exist or you do not have permission to view it."
  ],
  "errors": {}
}
```

Instead, there is a variant that uses *cf[ID]* (see the JIRA documentation), where *ID* is the id of the custom field, in our case *10006*. Note that the brackets must be URL-encoded, so *cf[10006]* turns into *cf%5B10006%5D*:

```
curl -u 'jirauser:jirapwd' \
'https://jira.example.com/rest/api/latest/search?jql=project=CSE2+and+cf%5B10006%5D!=NULL'|jq .
```

This means that the code for sync must detect whether a custom field is being used, extract the id, and use that in the query.

Chapter 4

Guides

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

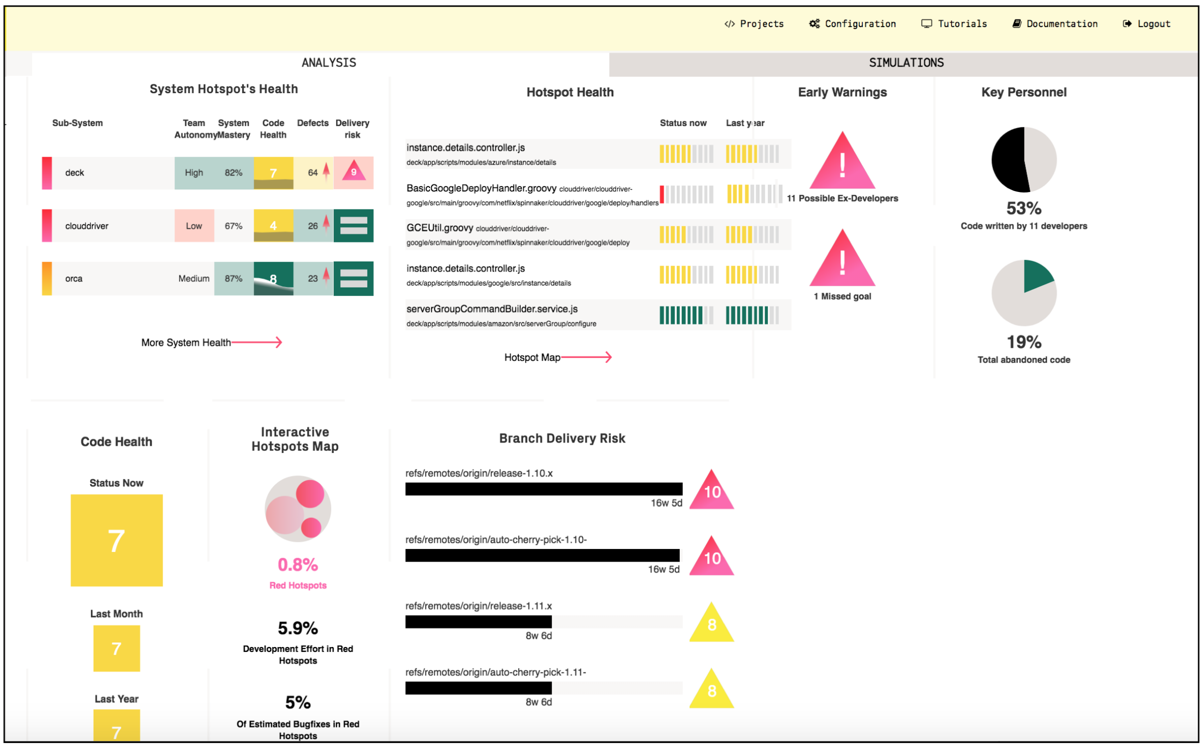
4.1 Dashboards

4.1.1 CodeScene's Dashboards: The Status of Your Codebase at a Glance

CodeScene comes with several dashboards, each one serving a specific use case. In this guide we introduce the dashboard for an analysis, the inter-product dashboard, as well as the system health dashboard.

The Analysis Dashboard Presents the main KPIs

Each analysis project gets its own dashboard where you see the high level results of each analysis.



Sub-System	Team Autonomy	System Mastery	Code Health	Defects	Delivery risk
deck	High	82%	7	64	9
clouddriver	Low	67%	4	26	
orca	Medium	87%	8	23	

instance.details.controller.js deck/app/scripts/modules/azure/instance/details
BasicGoogleDeployHandler.groovy clouddriver/clouddriver-google/src/main/groovy/com/netflix/spinnaker/clouddriver/google/deploy/handlers
GCEUtil.groovy clouddriver/clouddriver-google/src/main/groovy/com/netflix/spinnaker/clouddriver/google/deploy
instance.details.controller.js deck/app/scripts/modules/google/src/instance/details
serverGroupCommandBuilder.service.js deck/app/scripts/modules/amazon/src/serverGroup/configure

refs/remotes/origin/release-1.10.x 16w 5d 10
refs/remotes/origin/auto-cherry-pick-1.10- 16w 5d 10
refs/remotes/origin/release-1.11.x 8w 6d 8
refs/remotes/origin/auto-cherry-pick-1.11- 8w 6d 8

Fig. 4.1: CodeScene's interactive analysis dashboard presents the key metrics at a glance.

The dashboard presents the following information:

- *System Hotspot's Health*: Presents hotspots on an architectural level. The hotspots are sorted according to the development activity in each component/sub-system/service. Each hotspot has the following information:
  - *Team Autonomy* measures the degree of inter-team coordination in the code. The higher the team autonomy, the lower the risk for coordination issues and conflicting feature changes.
  - *System Mastery* measures the percentage of the code that the current development team has written. A low number indicates knowledge loss.
  - *Code Health* measures how easy and affordable it will be to understand, extend, and modify that part of the code. Code Health goes from *10* (code that's relatively easy to understand) to *1* which indicates severe maintenance problems.
  - *Defects* shows the number of bug fixes over the past month. The arrows indicate if the defects have increased or decreased compared to the previous month.
  - *Delivery Risk* is a prediction of recent high-risk changes that have been made to the code of a particular architectural component. Use this to focus code reviews and testing activities. The risk prediction algorithm is described in more detail in *Risk Analysis* (page 121).
- *Hotspot Health*: This table list the top five hotspots in the codebase, together with the trend score for their Code Health. These hotspots are typically the ones where code improvements have the largest pay off.
- *Early Warnings*: CodeScene auto-detects potential problems and present them here in a prioritized order. Click on one of the warnings to get more details and act on the findings.
- *Key Personnel*: Detects a key personnel exposure where a significant part of your code is devloped by few people. Follow-up this finding by invesigating the knowledge distribution in CodeScene.
- *Code Health* simply shows that trend over the past month and year. A decline in Code Health is a sign that the switch has to be changed from adding new features towards improving the code in the top hotspots.
- *Interactive Hotspots Map* let you explore your whole codebase and identify technical debt as described in /guides/usage/index.
- *Branch Delivery Risk* shows you the predicted delivery risk and, when clicked on, detailed statistics of active branches. Use this to focus extra testing to where it's likely to be needed the most.

Monitor a Specific Codebase via the System Health Dashboard

The System Health dashboad provides a real-time monitoring option. Use that one together with the full screen mode to display your system's high-level status on a TV screen.

The Inter-Project Dashboard gives an Overview of all your Codebases

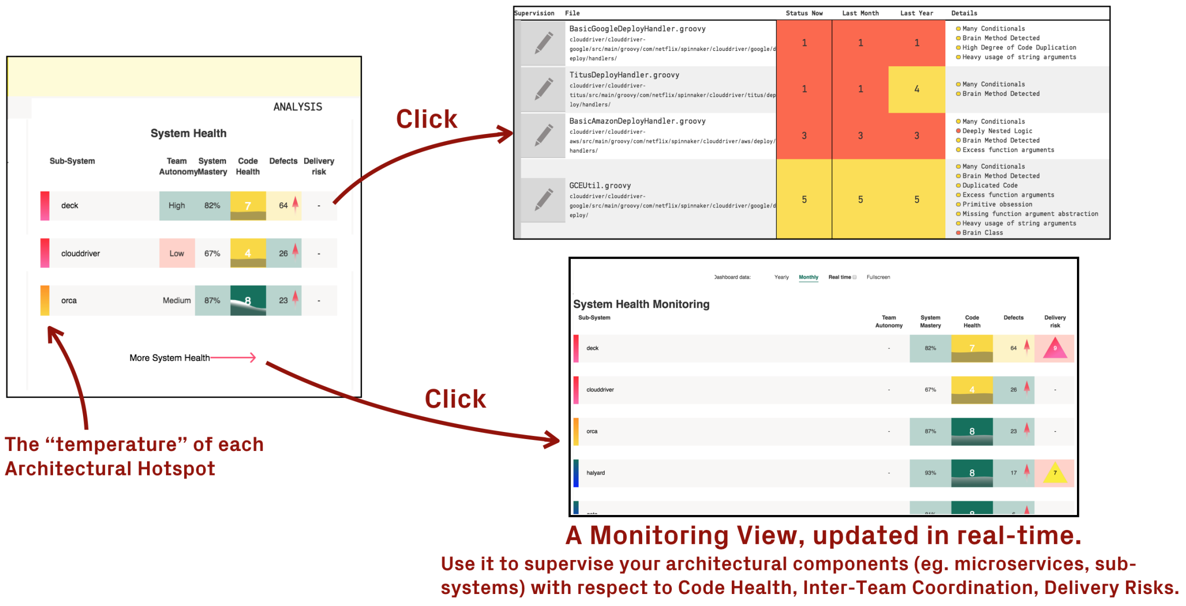
CodeScene also presents an inter-project dashboard that gives you a high-level overview of all your products, as shown in Fig. 2.8.

Using these key metrics on the dashboard, in particular the progress on the goals, point you the products/codebase that you need to inspect in more depth.

Finally, note that you can restrict access to certain analysis projects as described in *Users and Roles* (page 166).

4.2 Technical

4.2.1 Hotspots


ANALYSIS
Click
Click
More System Health
The “temperature” of each Architectural Hotspot
A Monitoring View, updated in real-time.
Use it to supervise your architectural components (eg. microservices, sub-systems) with respect to Code Health, Inter-Team Coordination, Delivery Risks.

System Health

Sub-System	Team Autonomy	System Mastery	Code Health	Defects	Delivery risk
deck	High	82%	7	64	-
clouddriver	Low	67%	4	26	-
orca	Medium	87%	8	23	-

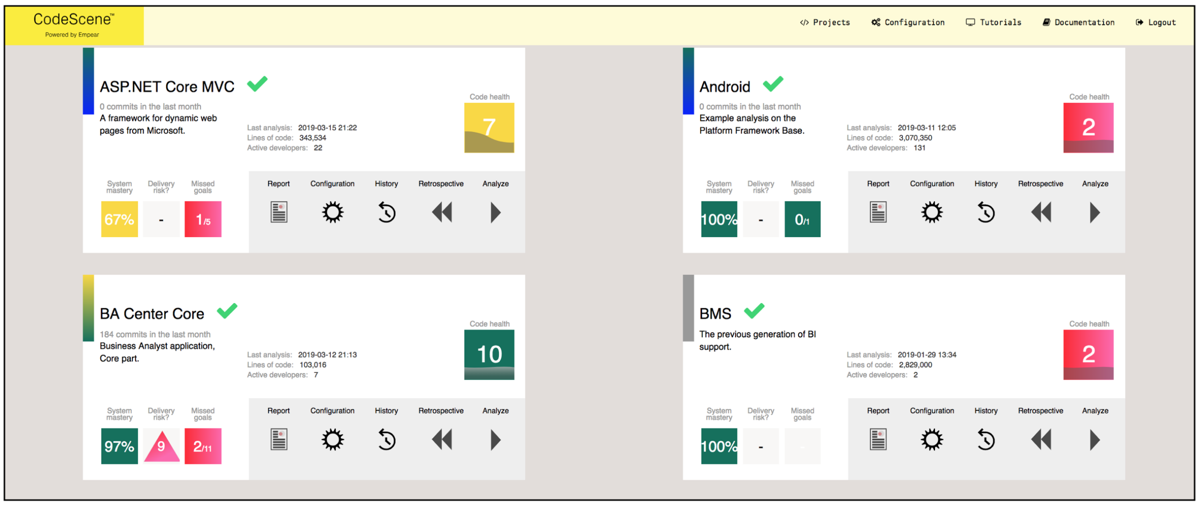
Supervision	File	Status Now	Last Month	Last Year	Details
	BasicGoogleDeployHandler.groovy clouddriver/clouddriver-google/src/main/groovy/com/netflix/spinnaker/clouddriver/google/deploy/handlers/	1	1	1	Many Conditionals; Brain Method Detected; High Degree of Code Duplication; Heavy usage of string arguments
	TitusDeployHandler.groovy clouddriver/clouddriver-titus/src/main/groovy/com/netflix/spinnaker/clouddriver/titus/deploy/handlers/	1	1	4	Many Conditionals; Brain Method Detected
	BasicAmazonDeployHandler.groovy clouddriver/clouddriver-aws/src/main/groovy/com/netflix/spinnaker/clouddriver/aws/deploy/handlers/	3	3	3	Many Conditionals; Deeply Nested Logic; Brain Method Detected; Excess function arguments
	GCEUtil.groovy clouddriver/clouddriver-google/src/main/groovy/com/netflix/spinnaker/clouddriver/google/deploy/	5	5	5	Many Conditionals; Brain Method Detected; Duplicated Code; Excess function arguments; Primitive obsession; Missing function argument abstraction; Heavy usage of string arguments; Brain Class

Dashboard data: Yearly Monthly Real time Fullscreen

System Health Monitoring

Sub-System	Team Autonomy	System Mastery	Code Health	Defects	Delivery risk
deck	-	82%	7	64	9
clouddriver	-	67%	4	26	-
orca	-	87%	8	23	-
halyard	-	93%	8	17	7

Fig. 4.2: The System Health dashboard lets you monitor the evolution of your sub-systems, services, or microservices.



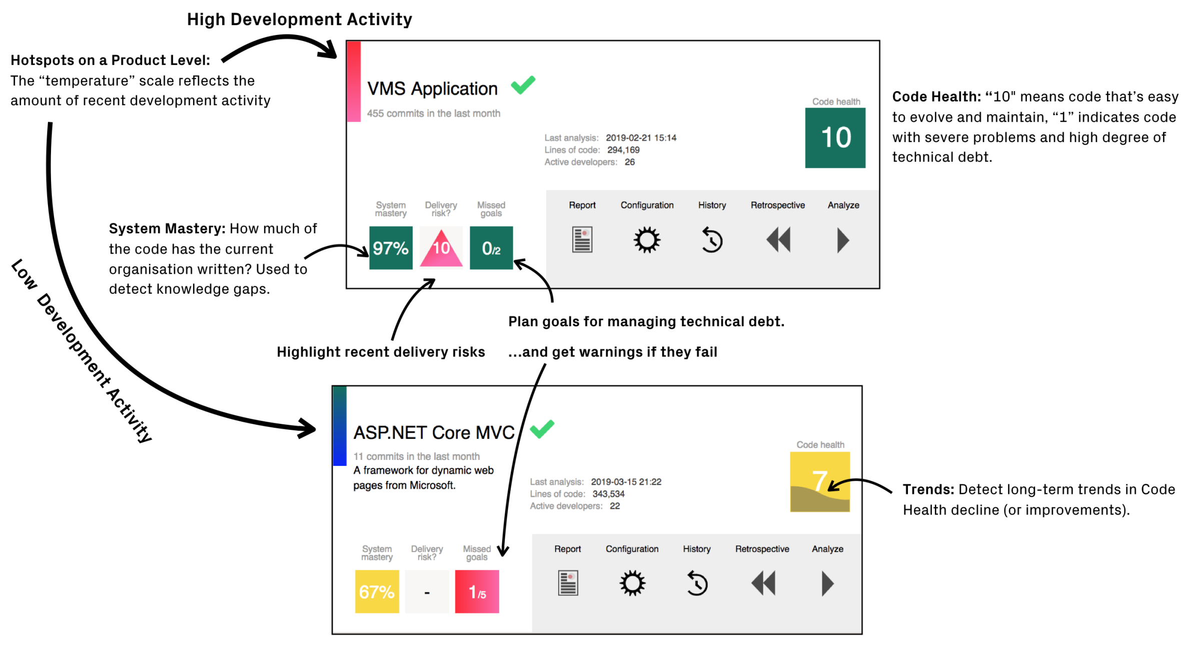
ASP.NET Core MVC
0 commits in the last month
A framework for dynamic web pages from Microsoft.
Last analysis: 2019-03-15 21:22
Lines of code: 343,534
Active developers: 22
Code health: 7
System mastery: 67% · Delivery risk?: - · Missed goals: 1/5

Android
0 commits in the last month
Example analysis on the Platform Framework Base.
Last analysis: 2019-03-11 12:05
Lines of code: 3,070,350
Active developers: 131
Code health: 2
System mastery: 100% · Delivery risk?: - · Missed goals: 0/1

BA Center Core
184 commits in the last month
Business Analyst application, Core part.
Last analysis: 2019-03-12 21:13
Lines of code: 103,016
Active developers: 7
Code health: 10
System mastery: 97% · Delivery risk?: 9 · Missed goals: 2/11

BMS
The previous generation of BI support.
Last analysis: 2019-01-29 13:34
Lines of code: 2,829,000
Active developers: 2
Code health: 2
System mastery: 100% · Delivery risk?: - · Missed goals: -

Fig. 4.3: The high-level dashboard shows that status of all your products at a glance.



Hotspots on a Product Level: The "temperature" scale reflects the amount of recent development activity

Code Health: "10" means code that's easy to evolve and maintain, "1" indicates code with severe problems and high degree of technical debt.

System Mastery: How much of the code has the current organisation written? Used to detect knowledge gaps.

Plan goals for managing technical debt.

Highlight recent delivery risks

...and get warnings if they fail

Trends: Detect long-term trends in Code Health decline (or improvements).

Fig. 4.4: The CodeScene inter-project dashboard explained.

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. 4.5. 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.

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

Explore the Hotspot Activity

CodeScene 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 an easy to explore, interactive visualization of your hotspots. Fig. 4.6 shows an example from the Visual Studio Code codebase.

The Hotspot visualization makes it easy to identify the parts of your code where most development effort is spent. In a larger codebase you want to let CodeScene identify your refactoring targets. Let's see how that's done.

Focus on your Refactoring Targets

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.

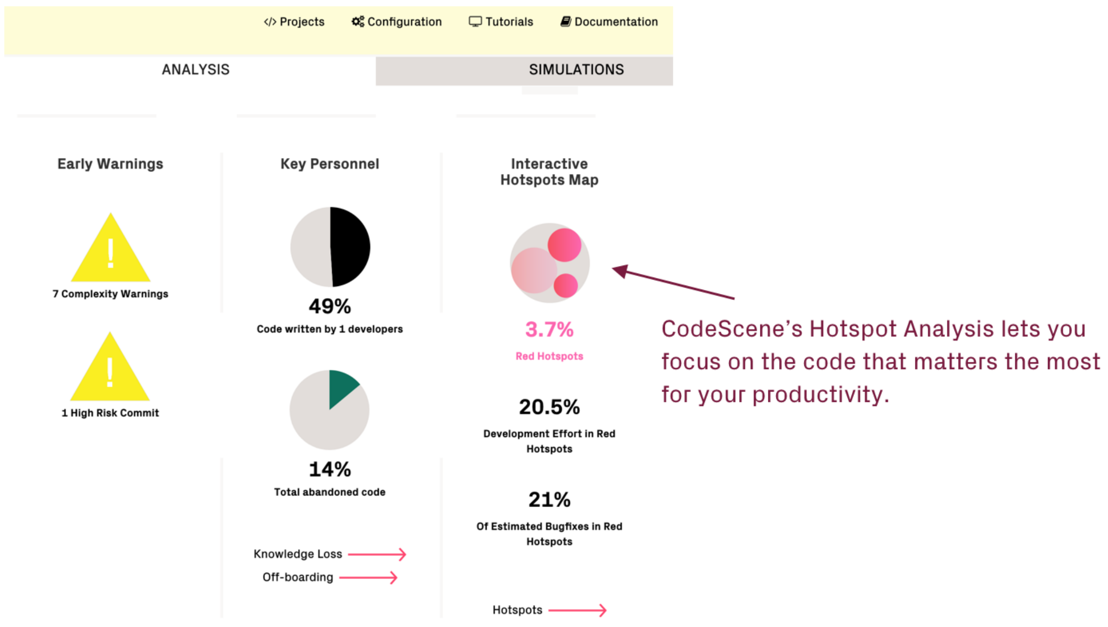


Fig. 4.5: The dashboard gives you a high-level overview of the Hotspot activity in your code.

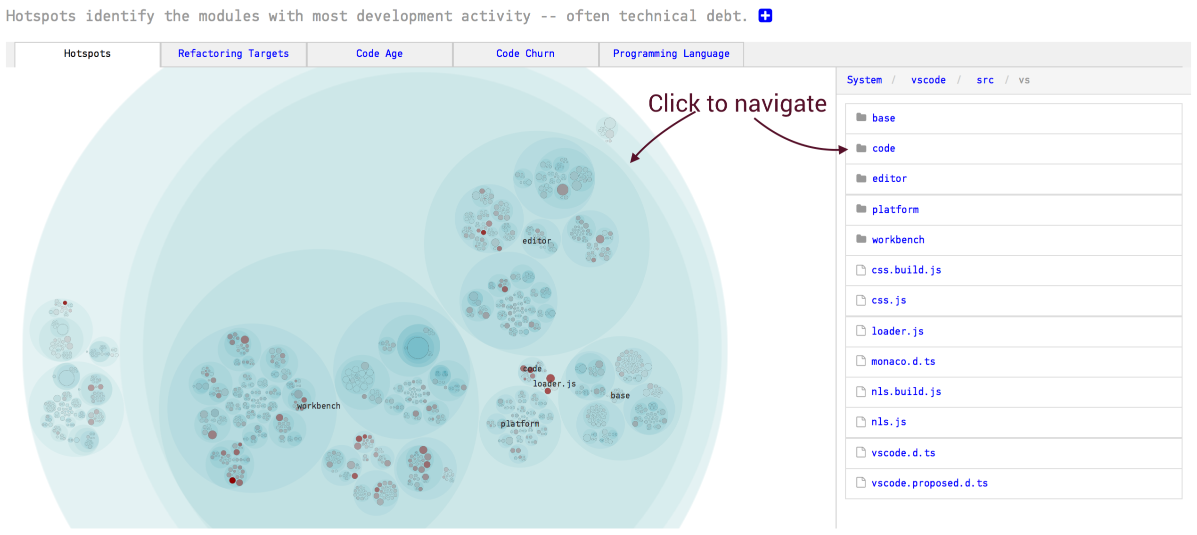

Hotspots identify the modules with most development activity -- often technical debt.

Fig. 4.6: Hotspots in the Visual Studio Code codebase.

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. 4.7.

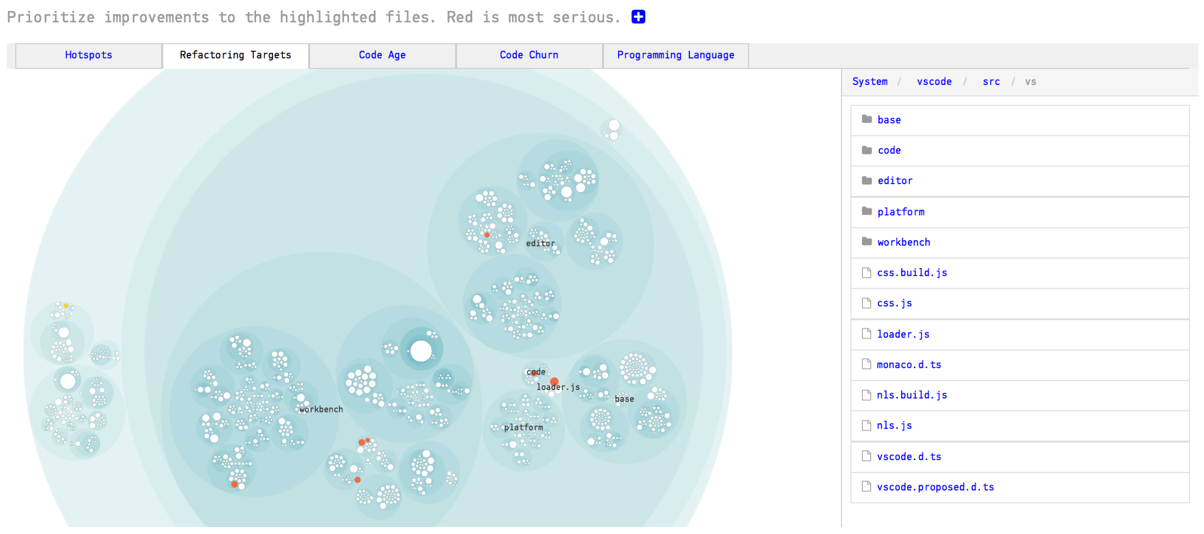
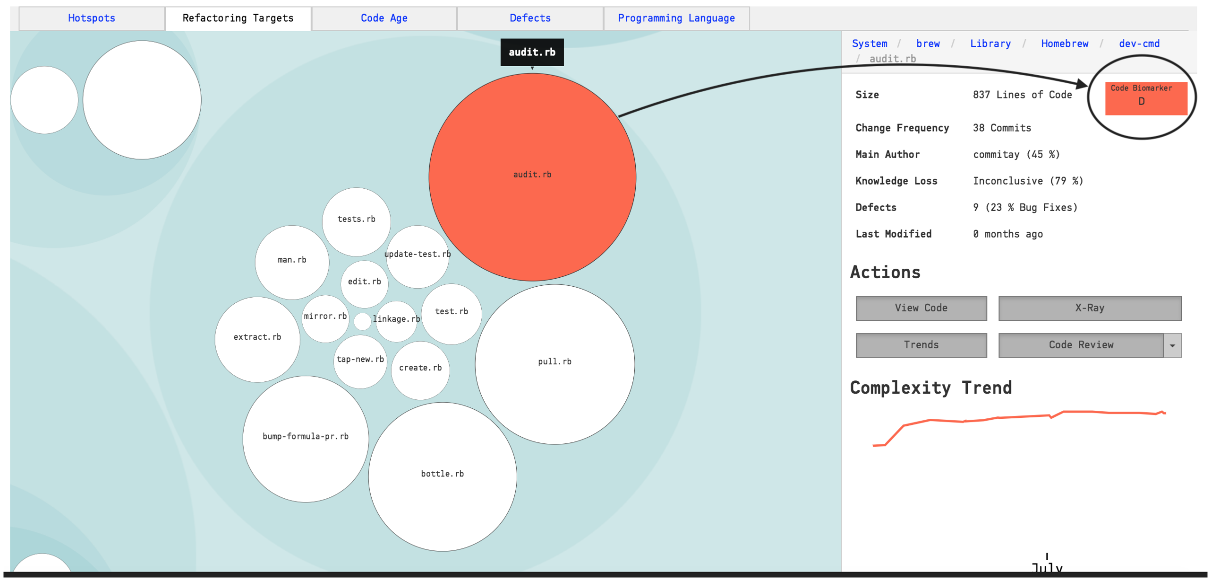


Fig. 4.7: CodeScene prioritizes the Hotspots in your code.

The red hotspots are the ones you want to focus your attention on; If there is any technical debt or code quality issues in the red hotspots, then improvements to those parts are likely to give you a real return on your investment.

Use the hotspot's biomarker score to get a quick assessment of potential technical debt or maintenance problems as shown in Fig. 4.8. We talk more about biomarkers in the next section.



Size	837 Lines of Code
Change Frequency	38 Commits
Main Author	commitay (45 %)
Knowledge Loss	Inconclusive (79 %)
Defects	9 (23 % Bug Fixes)
Last Modified	0 months ago

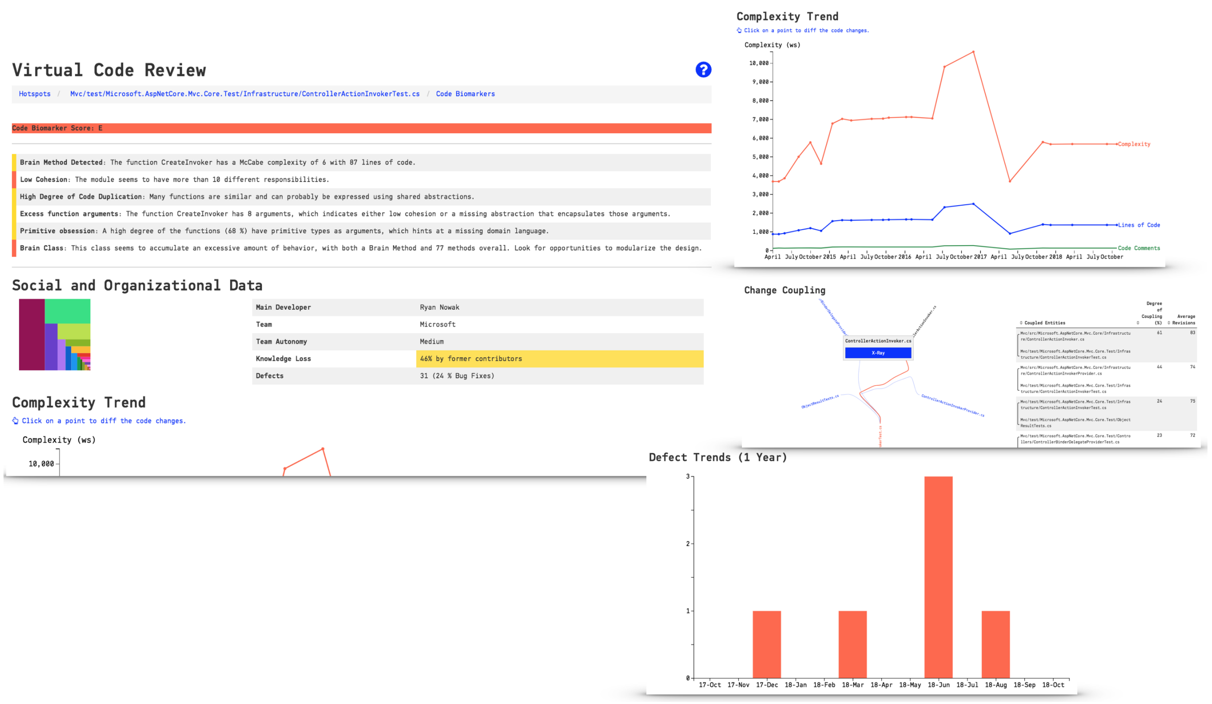
Fig. 4.8: CodeScene prioritizes the Hotspots in your code.

Once you've addressed those hotspots, the yellow hotspots become interesting as well. A yellow hotspot is secondary investigative target, albeit not as severe as the red category. Now, let's explore the biomarkers calculated for each hotspot.

Shrink the Problem Space with Main Suspects

The ranked hotspots presented as *Refactoring Targets* are 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.

Hence, CodeScene includes a virtual code reviewer for any file. The virtual code review will aggregate the most significant metrics for your chosen file, as seen in Fig. 4.9.



Virtual Code Review

Hotspots / Mvc/test/Microsoft.AspNetCore.Mvc.Core.Test/Infrastructure/ControllerActionInvokerTest.cs / Code Biomarkers

Code Biomarker Score: E

Brain Method Detected: The function CreateInvoker has a McCabe complexity of 6 with 87 lines of code.

Low Cohesion: The module seems to have more than 10 different responsibilities.

High Degree of Code Duplication: Many functions are similar and can probably be expressed using shared abstractions.

Excess function arguments: The function CreateInvoker has 8 arguments, which indicates either low cohesion or a missing abstraction that encapsulates those arguments.

Primitive obsession: A high degree of the functions (68 %) have primitive types as arguments, which hints at a missing domain language.

Brain Class: This class seems to accumulate an excessive amount of behavior, with both a Brain Method and 77 methods overall. Look for opportunities to modularize the design.

Social and Organizational Data

Main Developer	Ryan Nowak
Team	Microsoft
Team Autonomy	Medium
Knowledge Loss	46% by former contributors
Defects	31 (24 % Bug Fixes)

Fig. 4.9: Get a holistic overview of your hotspot.

Using CodeScene's *Code Biomarkers–A Virtual Code Reviewer aware of Code Health* (page 48), you're also able to get a quick classification on possible maintenance issues as shown in Fig. 4.10.

The main advantage of using hotspots to guide 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.

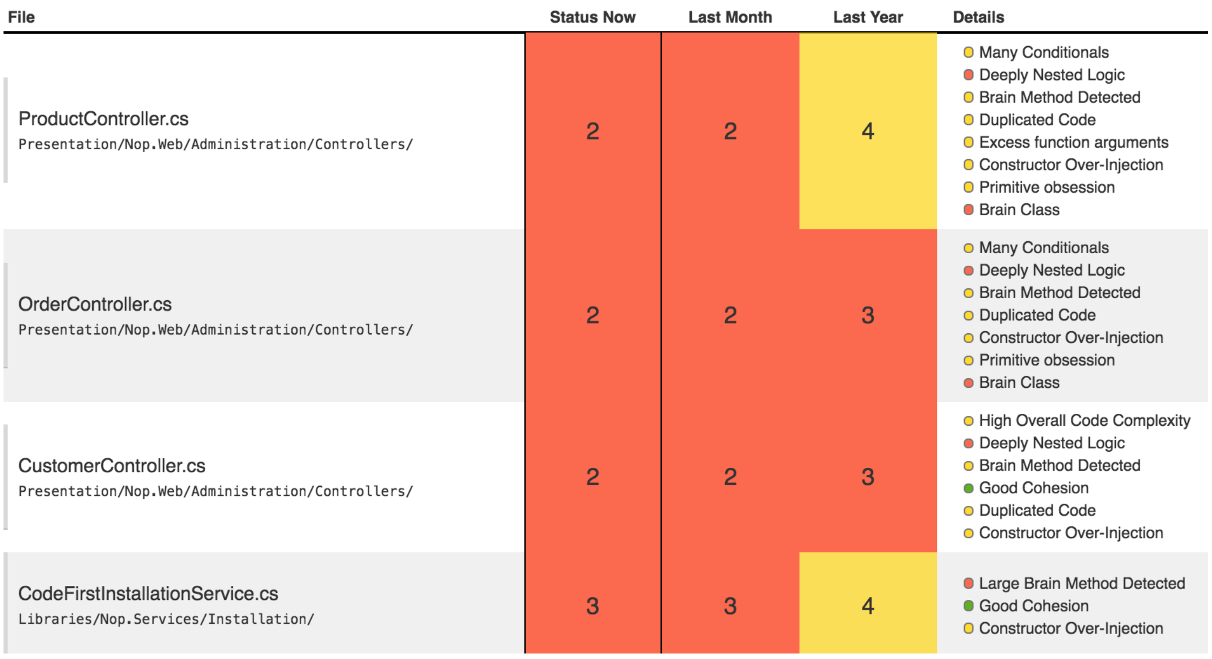
Use Defects to put Costs on Hotspots

When you come across hotspots with severe maintenance issues, there's always going to be a trade-off: do we pay-off the worst technical debt or should we continue to shoehorn yet another feature into the hotspot? Ideally, we would like to know for sure that if we invest, say, two weeks into refactoring the code, then that effort will pay-off immediately. At the time of writing, there's unfortunately no way of looking into the future. What we can do instead is to look at the existing costs and consequences of *not* doing any preventive and pro-active code improvements.

For this purpose, CodeScene comes with a *Defect Density* view. Since most organizations have a known and estimated number on how much a defect costs, let's use defects to predict the costs of any sub-optimal code we might find in our hotspots. Fig. 4.11 shows an example from CodeScene's dashboard.

The statistics on the dashboard tells us the following things about the development costs in our codebase:

- The prioritized hotspots only make up 5.5% of the total codebase, yet
- we spend 17.6% of our development efforts in those hotspots, and
- 23% of all bugs that we detect and fix are in that small part of the code.



File	Status Now	Last Month	Last Year	Details
ProductController.cs `Presentation/Nop.Web/Administration/Controllers/`	2	2	4	🟡 Many Conditionals 🔴 Deeply Nested Logic 🟡 Brain Method Detected 🟡 Duplicated Code 🟡 Excess function arguments 🟡 Constructor Over-Injection 🟡 Primitive obsession 🔴 Brain Class
OrderController.cs `Presentation/Nop.Web/Administration/Controllers/`	2	2	3	🟡 Many Conditionals 🔴 Deeply Nested Logic 🟡 Brain Method Detected 🟡 Duplicated Code 🟡 Constructor Over-Injection 🟡 Primitive obsession 🔴 Brain Class
CustomerController.cs `Presentation/Nop.Web/Administration/Controllers/`	2	2	3	🟡 High Overall Code Complexity 🔴 Deeply Nested Logic 🟡 Brain Method Detected 🟢 Good Cohesion 🟡 Duplicated Code 🟡 Constructor Over-Injection
CodeFirstInstallationService.cs `Libraries/Nop.Services/Installation/`	3	3	4	🔴 Large Brain Method Detected 🟢 Good Cohesion 🟡 Constructor Over-Injection

Fig. 4.10: The Code Biomarkers shows the status of your hotspots at a glance.

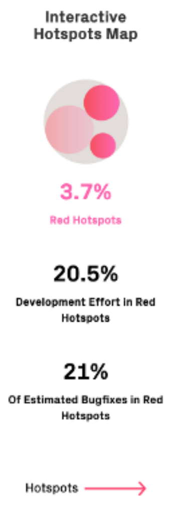

Interactive
Hotspots Map
3.7%
Red Hotspots
20.5%
Development Effort in Red Hotspots
21%
Of Estimated Bugfixes in Red Hotspots
Hotspots

Fig. 4.11: CodeScene's dashboard displays the bug density of the prioritized hotspots.

Before we move on, let's point out that 23% is actually on the lower-end; In most codebases, the top hotspots will be responsible for an even larger percentage of all fixed defects. This has direct implications on the costs of the whole system.

CodeScene's *Defect Density* view shows how distributed our bug fixes are, which lets you correlate defects with hotspots as shown in Fig. 4.12.

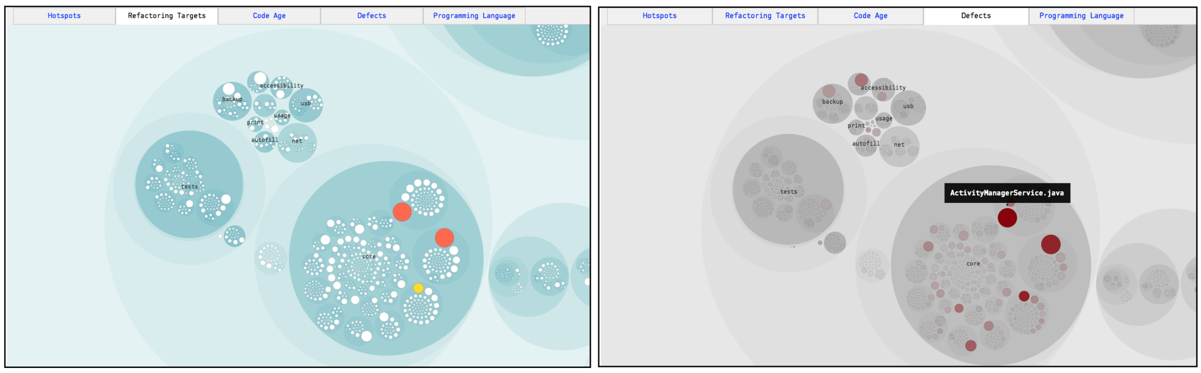
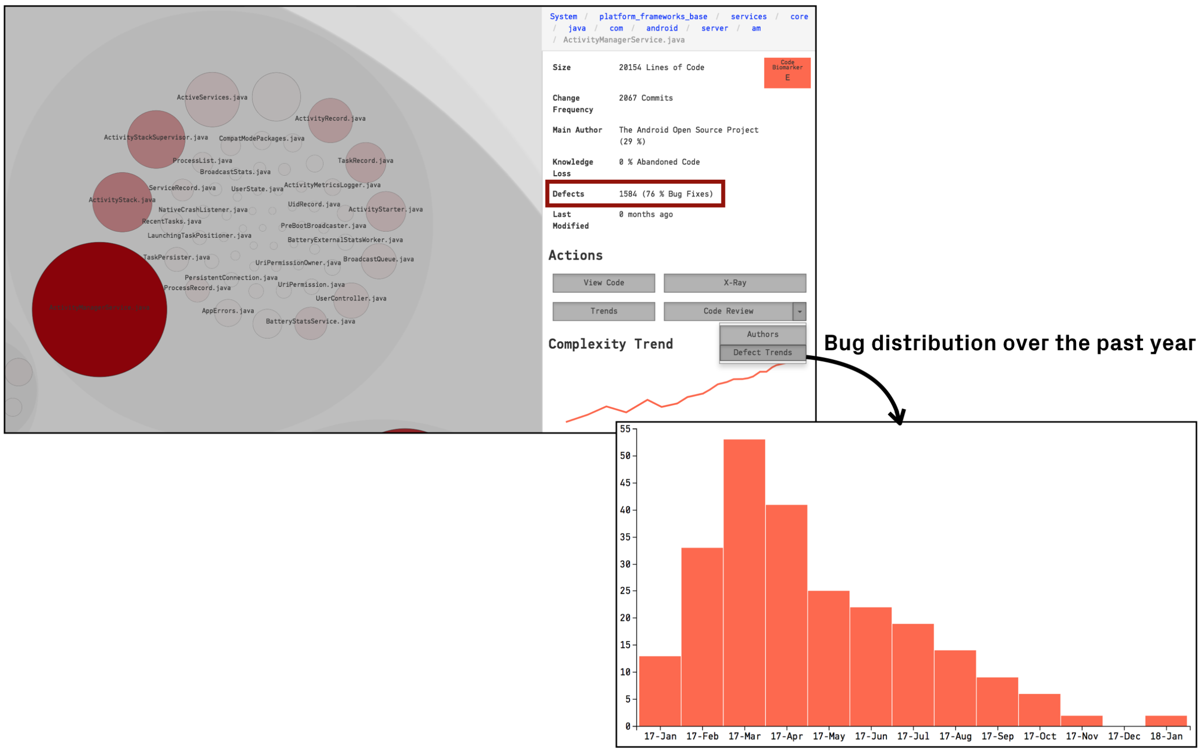


Fig. 4.12: Correlate prioritized hotspots with the distribution of defects in the codebase.

CodeScene also lets you inspect the distribution of those defects over time, as show in Fig. 4.13.



System / platform_frameworks_base / services / core / java / com / android / server / am / ActivityManagerService.java

Size	20154 Lines of Code
Change Frequency	2067 Commits
Main Author	The Android Open Source Project (29 %)
Knowledge Loss	0 % Abandoned Code
Defects	1584 (76 % Bug Fixes)
Last Modified	0 months ago

Fig. 4.13: The distribution of bug fixes over the past year.

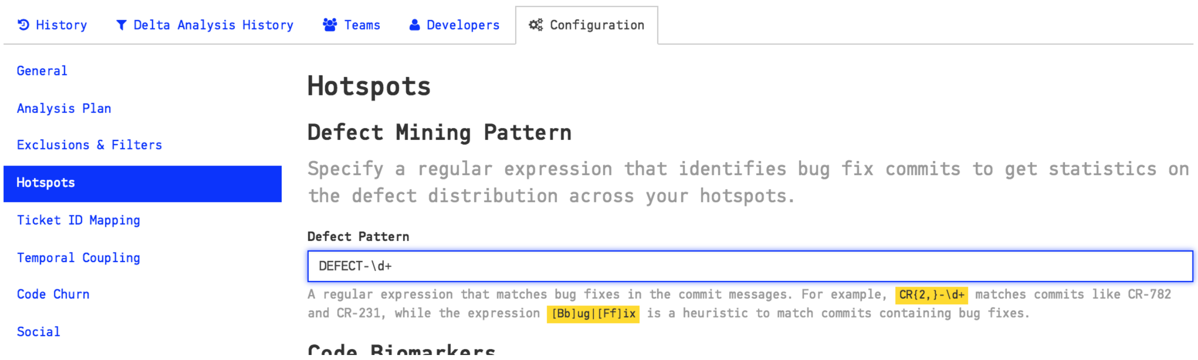
You use this information to detect if the hotspot seems to stabilize in terms of defects or if it's likely to be a growing problem.

Specify the Data Source for Defect Statistics

CodeScene needs a data source for its defect mining, and provides two different options depending on what data you have:

1. *Use Jira issues to identify defects*: This requires a that you integrate Jira with CodeScene. CodeScene will then use all issues identified as defects for its statistics. Specify this option in CodeScene's Jira configuration as described in *Integrate Jira Information into CodeScene* (page 29).
2. *Use commit message patterns to estimate defects*: If you have specific tags in your commit messages that can be used to identify defects, then this is a good option. As a fallback, CodeScene can use a heuristic that you can override that with

more specific patterns for higher precision, as shown in Fig. 4.14.



Hotspots

Defect Mining Pattern

Specify a regular expression that identifies bug fix commits to get statistics on the defect distribution across your hotspots.

Defect Pattern

```
DEFECT-\d+
```

A regular expression that matches bug fixes in the commit messages. For example, `CR{2,}-\d+` matches commits like CR-782 and CR-231, while the expression `[Bb]ug|[Ff]ix` is a heuristic to match commits containing bug fixes.

Code Biomarkers

Fig. 4.14: Configure a pattern to match defect information in your commit messages.

Dive into 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. 4.15.

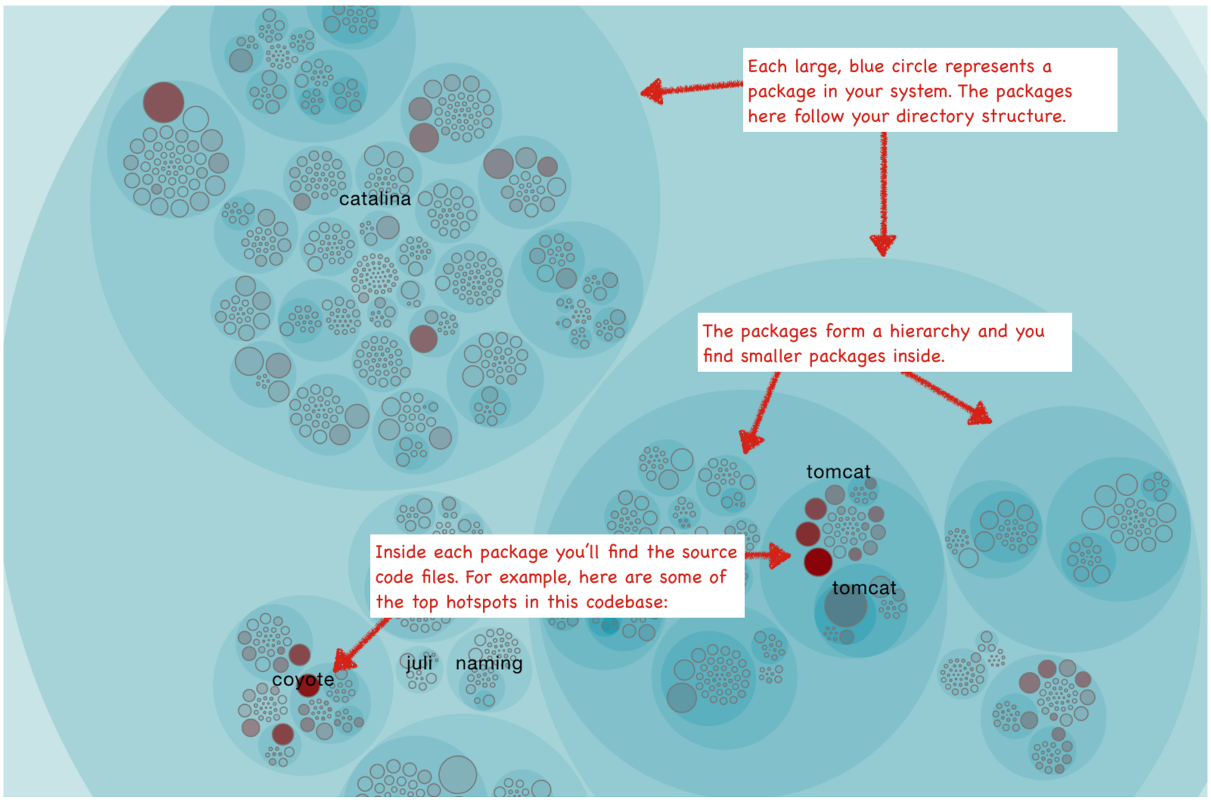
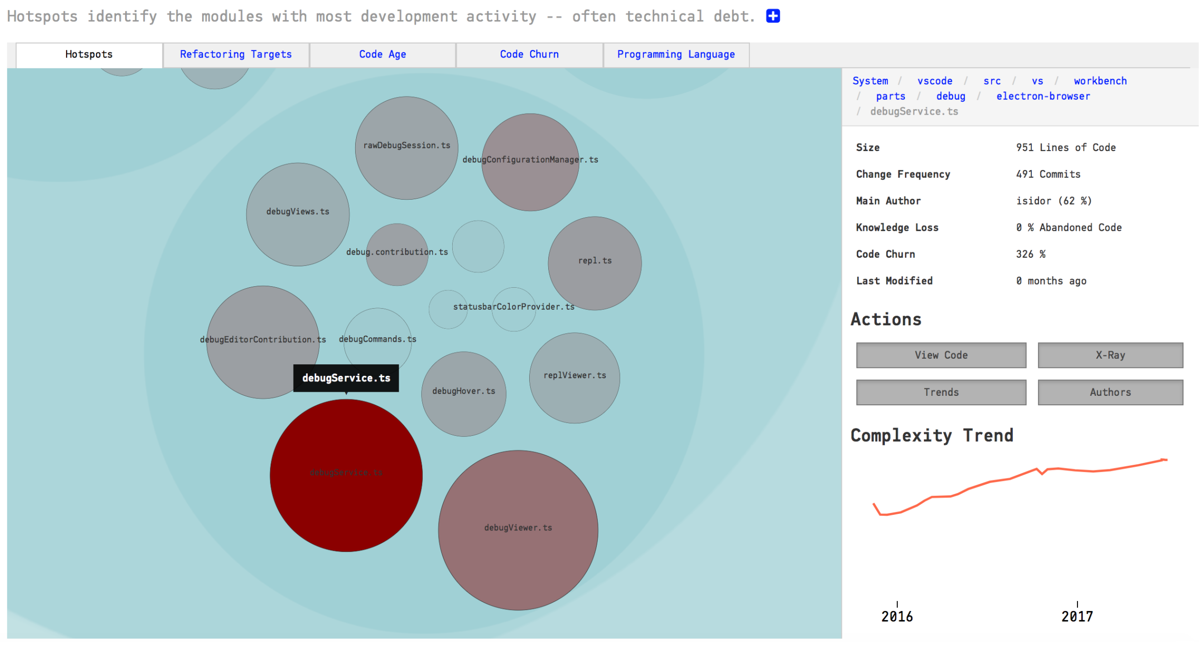

Each large, blue circle represents a package in your system. The packages here follow your directory structure.
The packages form a hierarchy and you find smaller packages inside.
Inside each package you'll find the source code files. For example, here are some of the top hotspots in this codebase:
catalina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

Fig. 4.15: Hotspots show you the activity in your codebase.

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. 4.16.


Hotspots identify the modules with most development activity -- often technical debt.
Hotspots
Refactoring Targets
Code Age
Code Churn
Programming Language
rawDebugSession.ts
debugConfigurationManager.ts
debugViews.ts
debug.contribution.ts
repl.ts
statusbarColorProvider.ts
debugEditorContribution.ts
debugCommands.ts
debugService.ts
debugHover.ts
replViewer.ts
debugViewer.ts
System / vscode / src / vs / workbench / parts / debug / electron-browser / debugService.ts
Actions
View Code
X-Ray
Trends
Authors
Complexity Trend
2016
2017

Size	951 Lines of Code
Change Frequency	491 Commits
Main Author	isidor (62 %)
Knowledge Loss	0 % Abandoned Code
Code Churn	326 %
Last Modified	0 months ago

Fig. 4.16: Click on a Hotspot to access the context menu.

Use the context menu to access the code for inspection, run CodeScene's X-Ray (see *X-Ray* (page 71)), investigate trends (see *Complexity Trends* (page 65)) and contributors (see *Parallel Development and Code Fragmentation* (page 112)).

CodeScene's hotspot view also lets you view different aspects of your system, as illustrated in Fig. 4.17.

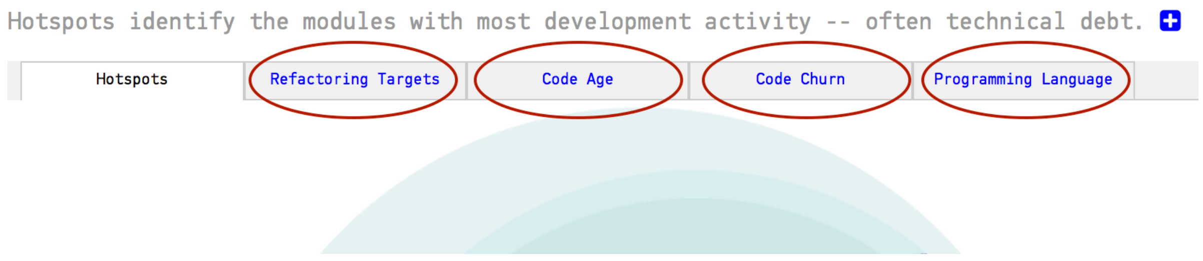

Hotspots identify the modules with most development activity -- often technical debt.
Hotspots
Refactoring Targets
Code Age
Code Churn
Programming Language

Fig. 4.17: Switch between different aspects in the hotspot view.

Just click on an aspect to view its data. For example, Fig. 4.18 shows the distribution of programming languages used in the implementation of a system.

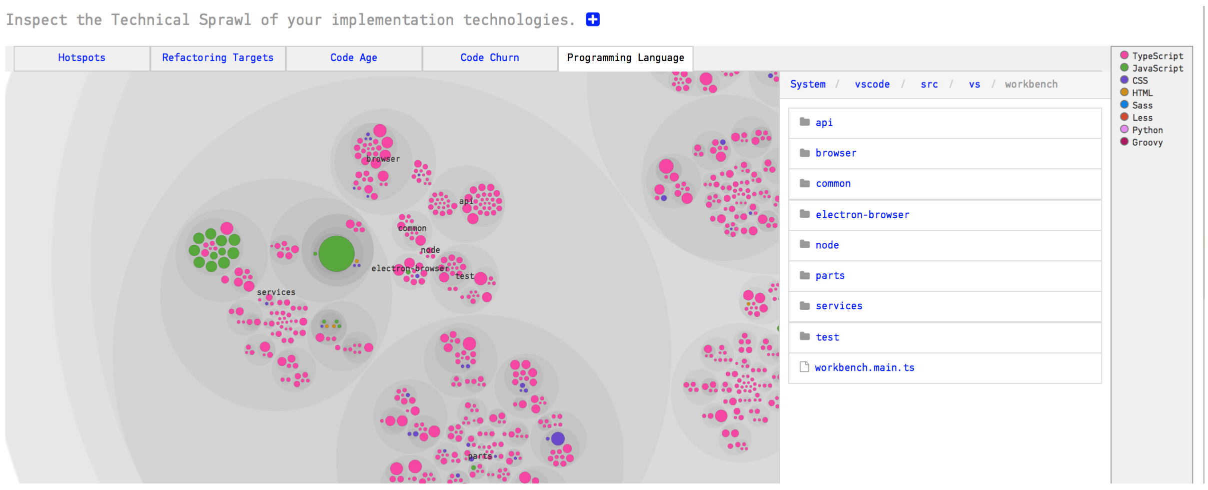


Fig. 4.18: The programming language aspect shows the technical sprawl in your codebase.

Use Code Churn as an Alternative Hotspot Metric

Another interesting aspect is *Code Churn*. 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.

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.

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.
- *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

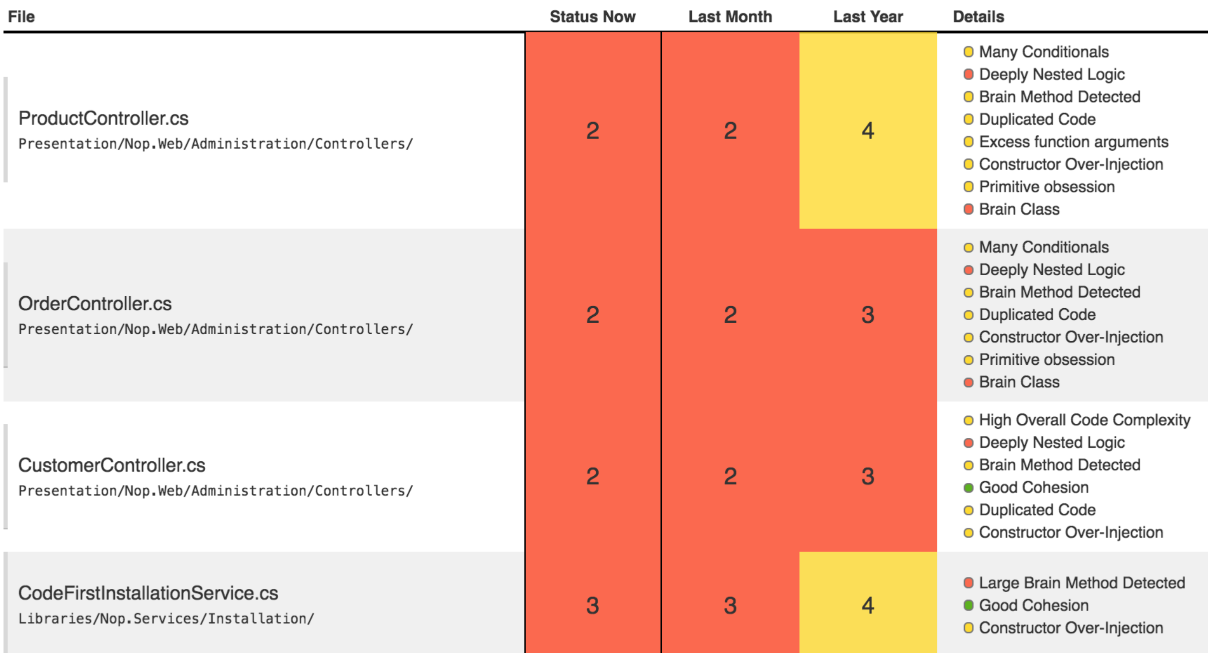
How well does Hotspots work in practice? Well, it turns out there's strong scientific support behind the metric. The research has often focused on bug predictions, which is relevant since bugs are one of the main issues behind expensive software maintenance.

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?

4.2.2 Code Biomarkers–A Virtual Code Reviewer aware of Code Health

In medicine, a biomarker is a measure that might indicate a particular disease or physiological state of an organism. CodeScene's biomarkers does the same for code. Combined with biomarker trends, this gives you a high level summary on the state of your hotspots and the direction your code is moving in.



File	Status Now	Last Month	Last Year	Details
ProductController.cs `Presentation/Nop.Web/Administration/Controllers/`	2	2	4	🟡 Many Conditionals 🔴 Deeply Nested Logic 🟡 Brain Method Detected 🟡 Duplicated Code 🟡 Excess function arguments 🟡 Constructor Over-Injection 🟡 Primitive obsession 🔴 Brain Class
OrderController.cs `Presentation/Nop.Web/Administration/Controllers/`	2	2	3	🟡 Many Conditionals 🔴 Deeply Nested Logic 🟡 Brain Method Detected 🟡 Duplicated Code 🟡 Constructor Over-Injection 🟡 Primitive obsession 🔴 Brain Class
CustomerController.cs `Presentation/Nop.Web/Administration/Controllers/`	2	2	3	🟡 High Overall Code Complexity 🔴 Deeply Nested Logic 🟡 Brain Method Detected 🟢 Good Cohesion 🟡 Duplicated Code 🟡 Constructor Over-Injection
CodeFirstInstallationService.cs `Libraries/Nop.Services/Installation/`	3	3	4	🔴 Large Brain Method Detected 🟢 Good Cohesion 🟡 Constructor Over-Injection

Fig. 4.19: The Code Biomarkers shows the status of your hotspots at a glance.

Biomarkers Indicate Code Health

The code biomarkers form the basis for CodeScene's **Code Health** metric that you see in the preceding figure. The Code Health metric is a score that goes from *10* (healthy code that relatively easy to understand and evolve) down to *1*, which indicates code with severe quality issues.

The code biomarkers are calculated from a combination of both properties of the code, as well as organizational factors behind the code that impacts the code health. In total, CodeScene calculates 25-30 code biomarkers depending on programming language. Examples include – but are not limited to – the following:

- **Brain Method**: A single function/method that centers too much behavior and becomes a local hotspot.
- **Nested Complexity**: This is typically revealed as *if*-statements inside other *if*-statments and/or loops, and is a construct that increases the risk for defects significantly.
- **Developer Congestion**: Code becomes a coordination bottleneck when multiple developers need to work on it in parallel (see *Parallel Development and Code Fragmentation* (page 112)).
- **Knowledge Loss due to former contributors**: If the developer behind a hotspot with low code healt leaves the organization, the maintenance risk increases significantly.
- **DRY (Don't Repeat Yourself) Violations**: CodeScene detects duplicated logic that is actually changed together in predictable patterns.

- **Primitive Obsession**: Code that uses a high degree of built-in, primitives such as *integers*, *strings*, *floats*, often lacks a domain language that encapsulates the validation and semantics of function arguments.

In combination with the Code Health score, CodeScene's biomarkers are like an extra, virtual team member that constantly reviews your code. Let's look into the biomarkers.

The Ideas Behind Code Biomarkers

We at Empear make heavy use of CodeScene ourselves. We use the tool as part of our services. Over the past years we have analyzed hundreds of different codebases, and there are some patterns that we have seen repeated over and over again. Thus, we started to implement support in CodeScene to auto-detect those patterns, and we called the feature biomarkers.

The biomarkers name requires a brief explanation. In general, we wanted to avoid terms like "quality" or "maintainability" since they are easy to game and, more serious, suggest an absolute truth. Instead we find that it's the trend that's most important: is the code evolving in the desired direction? In addition, an algorithm, no matter how smart, can only take us so far; at some level we want a human in the loop, and the code biomarkers are there to support that human by priming them on what to look for in the specific hotspot. Let's look at some examples.

Explore your Code's Biomarkers

If CodeScene has biomarker support for your language (see *X-Ray* (page 71) for a list of supported languages), you will get a high-level trend on your dashboard as shown in Fig. 4.20.

The Code Health scores on the dashboard show the (weighted) aggregation of the health of your prioritized hotspots. That is, the code that is most likely to drive the costs of new features and improvements. In this example, we see that a codebase that seems to degrade rapidly, going from a healthy score of *8* down to a low health of *4* in less than a year. That trend is a clear warning sign that the organization needs to invest more time into refactoring and code improvements.

Biomarkers Present Actionable Metrics

Before we move on, how do we know that the biomarkers and scores are relevant? Well, the biomarkers are built on top of CodeScene's other metrics and behavioral data. That means we only score the prioritized parts of the codebase, the one's that are most likely to impact development and maintenance costs as show in Fig. 4.21.

Using this principle, Code Biomarkers fill a number of important gaps:

- *Bridge the gap between developers and non-technical stakeholders*: The biomarkers visualization provides information to managers that help decide on when to take a step back, invest in technical improvements, and measure the effects.
- *Get immediate feedback on improvements*: The biomarker trends gives you immediate and visual feedback on the investments you do in refactorings.
- *Share an objective picture of your code quality*: The biomarker scores are based on baseline data from throusands of codebases, and your code is scored against an industry average of similar codebases.
- *Get suggestions on where to start refactorings*: The code biomarkers hint at specific problems in each file, which also suggests which refactorings that could be used to address the findings.

Let's demonstrate those properties by having a more detailed look at biomarkers in Fig. 4.22.

The biomarkers in `biomarkers-trend-examplef` provide detailed indications for each prioritized hotspot. We note that the top hotspot, *StageBuilder* has declined in health over the past months. We also note the warning sign for *DeltaHighlight.java*, which has degraded from a full *10* score to a medium health of *6*.

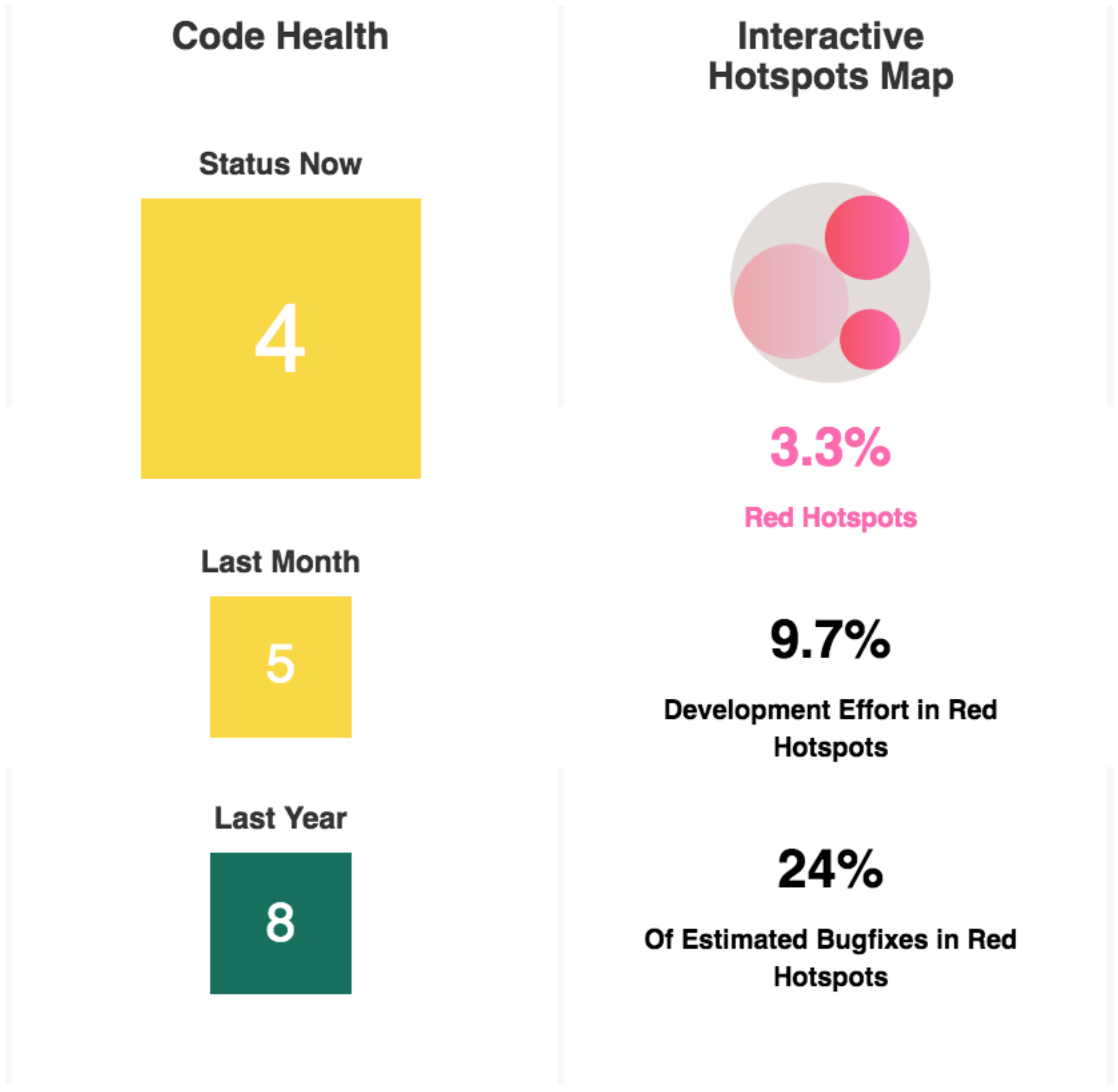


Fig. 4.20: Code Biomarkers summary on the analysis dashboard.

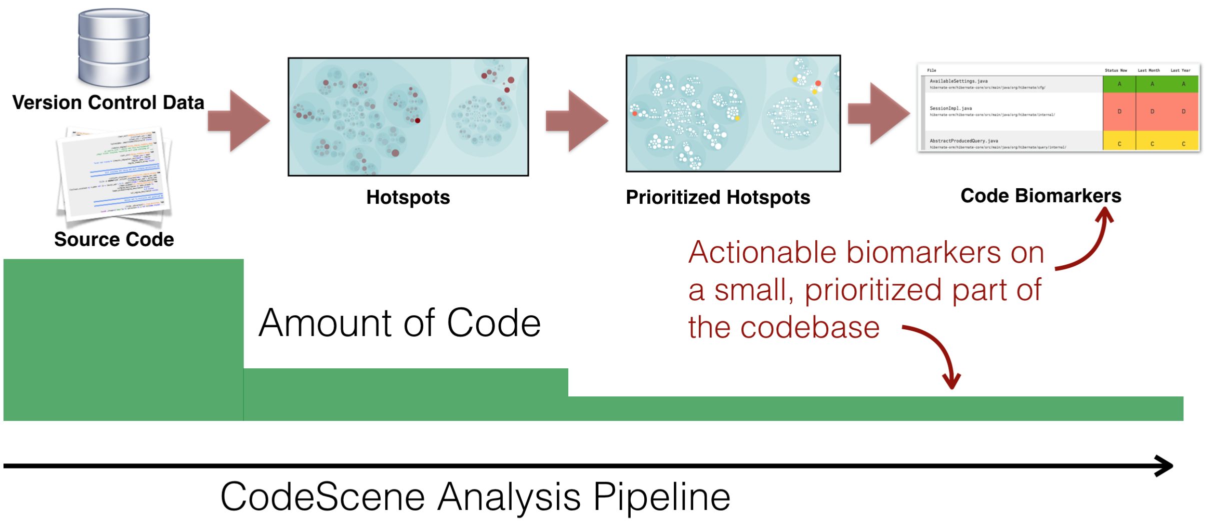
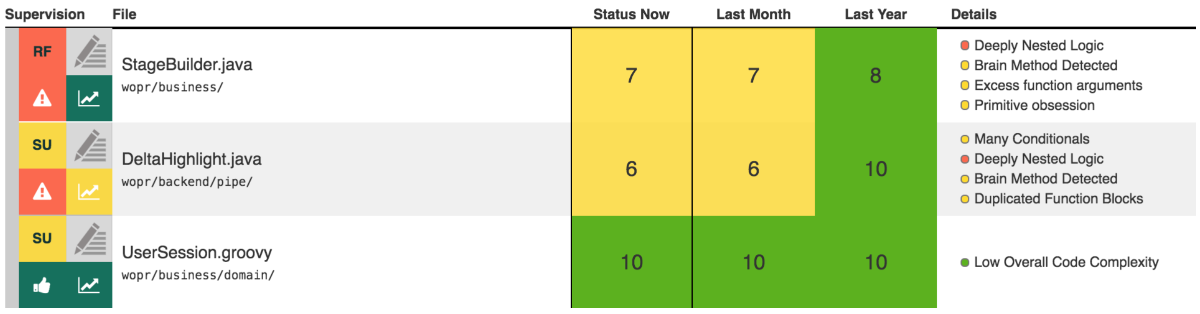

Version Control Data
Source Code
Hotspots
Prioritized Hotspots
Code Biomarkers
Actionable biomarkers on a small, prioritized part of the codebase
Amount of Code
CodeScene Analysis Pipeline

Fig. 4.21: Biomarkers are built on top of CodeScene's prioritized hotspots.



Supervision	File	Status Now	Last Month	Last Year	Details
RF	StageBuilder.java wopr/business/	7	7	8	● Deeply Nested Logic ● Brain Method Detected ● Excess function arguments ● Primitive obsession
SU	DeltaHighlight.java wopr/backend/pipe/	6	6	10	● Many Conditionals ● Deeply Nested Logic ● Brain Method Detected ● Duplicated Function Blocks
SU	UserSession.groovy wopr/business/domain/	10	10	10	● Low Overall Code Complexity

Fig. 4.22: Code Health Trends and Biomarkers for a specific project.

Note that the organization has already planned a set of goals for the hotspots in Fig. 4.20 using CodeScene's **Intelligent Notes** (see *Augmented Analysis with Intelligent Notes* (page 54)). This is important as the goal oriented workflow that CodeScene encourages, lets you track and manage technical debt in the context of your decisions and plans. Make sure to read the tutorial in *Hands On Behavioral Code Analysis: CodeScene Use Cases* (page 19) to ensure you get the most out of CodeScene.

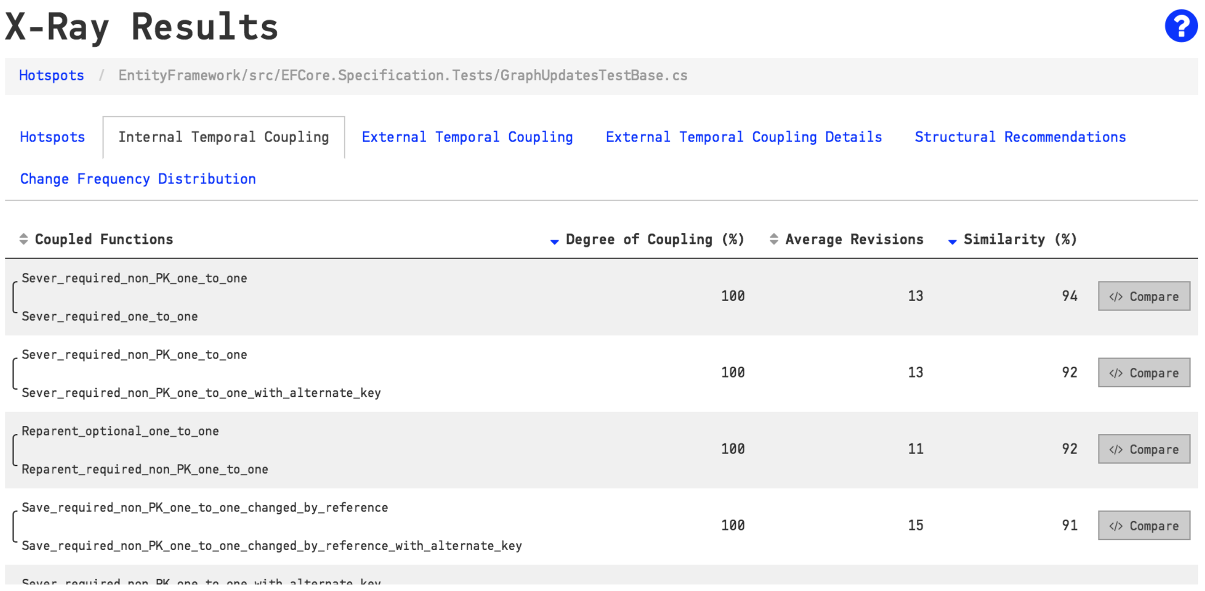
Finally, we get more details when we request the virtual code review via the lab bottle next to each hotspot. Use the virtual code review as an initial step towards refactoring decaying hotspots.

Guide Refactorings via the High-Level Patterns Presented in the Virtual Code Reviewer

Often, hotspots with low code health will contain brain methods and be low on cohesion. This is an indication that the hotspot lacks modularity on both function and class level. The code biomarkers will detect this and inform you.

Follow up with an X-Ray in order to prioritize local improvements based on how likely those refactoring are to pay off.

Using X-Ray, we can also investigate the code duplication reported by a biomarker. Duplicated code usually hints at one–or more–missing abstractions that we could introduce. Hence, we recommend to run an X-Ray analysis on the file to get more insights now that we know what to look for. We show an example of a *QueryTestBase.cs* X-Ray in Fig. 4.23.



X-Ray Results

Hotspots / EntityFramework/src/EFCore.Specification.Tests/GraphUpdatesTestBase.cs

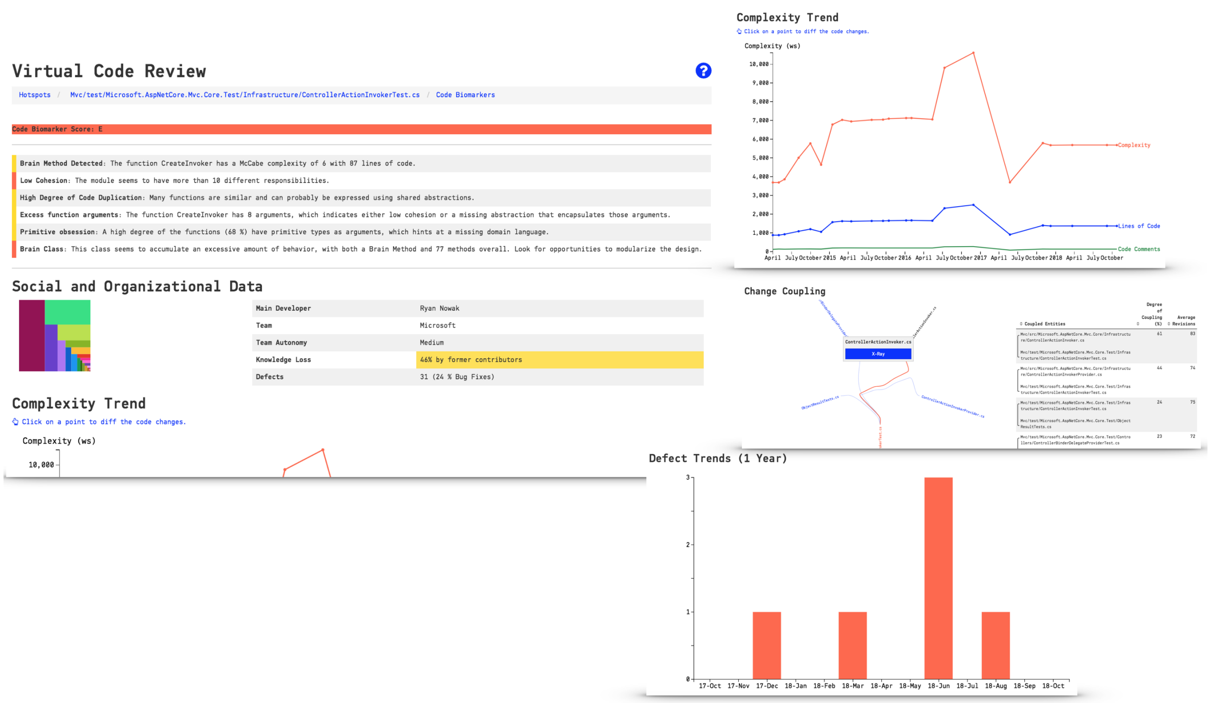
Hotspots | Internal Temporal Coupling | External Temporal Coupling | External Temporal Coupling Details | Structural Recommendations | Change Frequency Distribution

Coupled Functions	Degree of Coupling (%)	Average Revisions	Similarity (%)	
Sever_required_non_PK_one_to_one Sever_required_one_to_one	100	13	94	Compare
Sever_required_non_PK_one_to_one Sever_required_non_PK_one_to_one_with_alternate_key	100	13	92	Compare
Reparent_optional_one_to_one Reparent_required_non_PK_one_to_one	100	11	92	Compare
Save_required_non_PK_one_to_one_changed_by_reference Save_required_non_PK_one_to_one_changed_by_reference_with_alternate_key	100	15	91	Compare

Fig. 4.23: Use X-Ray to follow-up on the biomarkers.

Launch the Virtual Code Reviewer to Get a Holistic View of Hotspots

You can launch a virtual code reviewer for any file. The virtual code review will aggregate the most significant metrics for your chosen file, as seen in Fig. 4.9.



Virtual Code Review

Hotspots / Mvc/test/Microsoft.AspNetCore.Mvc.Core.Test/Infrastructure/ControllerActionInvokerTest.cs / Code Biomarkers

Code Biomarker Score: E

Brain Method Detected: The function CreateInvoker has a McCabe complexity of 6 with 87 lines of code.

Low Cohesion: The module seems to have more than 10 different responsibilities.

High Degree of Code Duplication: Many functions are similar and can probably be expressed using shared abstractions.

Excess function arguments: The function CreateInvoker has 8 arguments, which indicates either low cohesion or a missing abstraction that encapsulates those arguments.

Primitive obsession: A high degree of the functions (68 %) have primitive types as arguments, which hints at a missing domain language.

Brain Class: This class seems to accumulate an excessive amount of behavior, with both a Brain Method and 77 methods overall. Look for opportunities to modularize the design.

Social and Organizational Data

Main Developer	Ryan Nowak
Team	Microsoft
Team Autonomy	Medium
Knowledge Loss	46% by former contributors
Defects	31 (24 % Bug Fixes)

Fig. 4.24: *Get a holistic overview of your hotspot.*

The virtual code reviewer combines the social and technical analysis data you need to assess the severity of the biomarker findings:

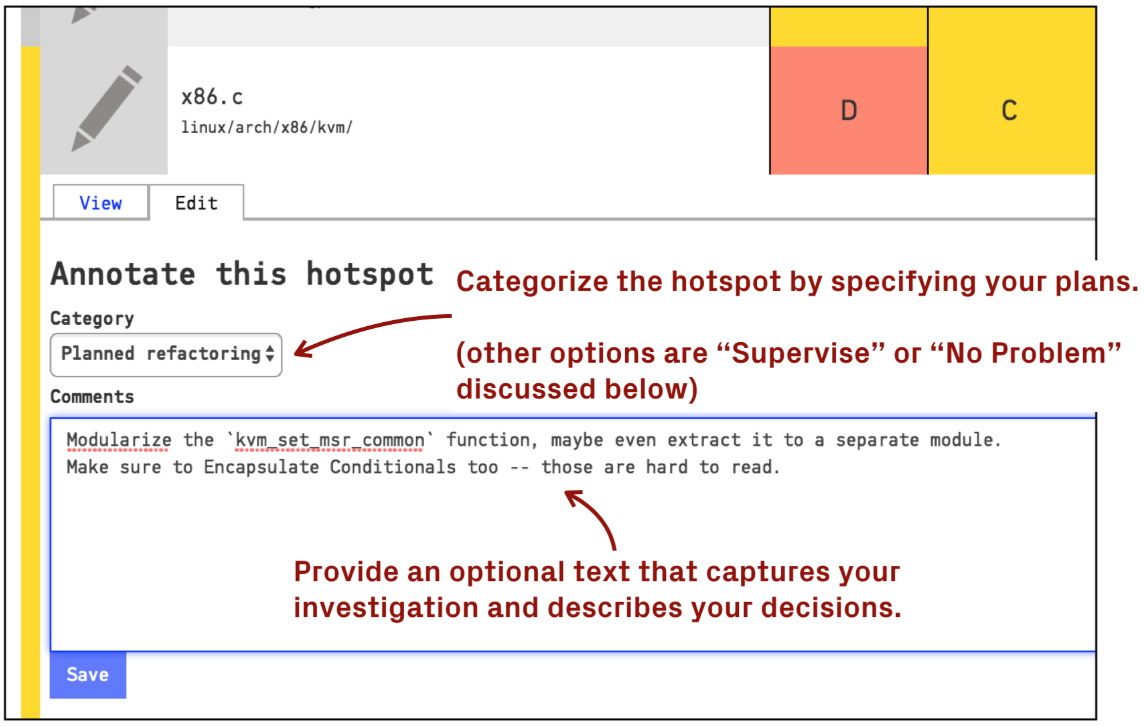
- Review the detailed biomarker indications to spot maintenance and quality issues.
- Detect potential inter-team coordination bottlenecks that should drive refactorings through the social metrics such as the team autonomy measure.
- See if it's a growing problem in the Complexity Trend.
- Investigate the change coupling, filtered for your selected file under review.
- Use the defect trend to estimate the cost of any technical or social debt you might find in the review.

Augment Hotspots to let CodeScene Supervise them

Once you have inspected a hotspot you can augmented the analysis with your observations. An augmented analysis lets you categorize your findings so that CodeScene can supervise and guide you based on the technical debt you identify. This augmented analysis provides a complete framework for managing technical debt and is described in detail in *Augmented Analysis with Intelligent Notes* (page 54)

Display the Biomarkers Monitor

CodeScene presents an additional monitor view where the biomarkers are continuously updated with the status of your ongoing work. Present the view on a TV in the office and use the information to communicate a shared understanding on the state of the codebase as shown in Fig. 4.26.



x86.c
linux/arch/x86/kvm/

Annotate this hotspot

Category: Planned refactoring

Comments:

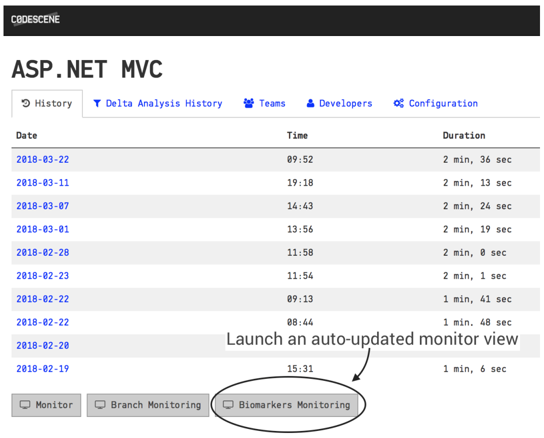
Modularize the `kvm_set_msr_common` function, maybe even extract it to a separate module.
Make sure to Encapsulate Conditionals too -- those are hard to read.

Categorize the hotspot by specifying your plans.

(other options are “Supervise” or “No Problem” discussed below)

Provide an optional text that captures your investigation and describes your decisions.

Fig. 4.25: Add an intelligent note to any hotspot in the Code Biomarkers view.



ASP.NET MVC

Date	Time	Duration
2018-03-22	09:52	2 min, 36 sec
2018-03-11	19:18	2 min, 13 sec
2018-03-07	14:43	2 min, 24 sec
2018-03-01	13:56	2 min, 19 sec
2018-02-28	11:58	2 min, 0 sec
2018-02-23	11:54	2 min, 1 sec
2018-02-22	09:13	1 min, 41 sec
2018-02-22	08:44	1 min. 48 sec
2018-02-20		
2018-02-19	15:31	1 min, 6 sec

Launch an auto-updated monitor view

Fig. 4.26: Display an always up-to-date view of your biomarkers.

Auto-Detect Degrading Biomarkers with Continuous Integration

CodeScene's delta analysis lets you supervise your code health as part of a continuous integration pipeline. This lets you auto-detect files that seem to degrade in quality through issues introduced in the current commit or pull request. See *Use a Delta Analysis to Save Time in Code Reviews* (page 125) for more details.

4.2.3 Augmented Analysis with Intelligent Notes

An augmented analysis lets you add contextual information to the analysis data as intelligent notes. In combination with hotspot analyses, augmented analysis provides a complete framework for managing technical debt, from detection to action.

Specify Contextual Information

Context matters:

- What features do you plan to implement next and what parts of the system will they affect? If you know a particular module requires an extension, you might want to start by a set of pro-active refactorings to make the new feature cheaper and less risky to implement.
- There's always a trade-off between adding new features versus improving existing code. So how much can you spend on re-work and improvements?
- Finally, there's a decision to be made on when is a piece of code is good enough.

To account for these scenarios, CodeScene introduces the concept of augmented code analysis. An augmented analysis lets you specify contextual information which is then processed as part of the analysis. This means that the analyses are aware of your goals and can measure the progress towards them. Let's get started with an example.

Augment Hotspots With Intelligent Notes

An augmented analysis is based on *intelligent notes*. CodeScene supports three different categories of intelligent notes:

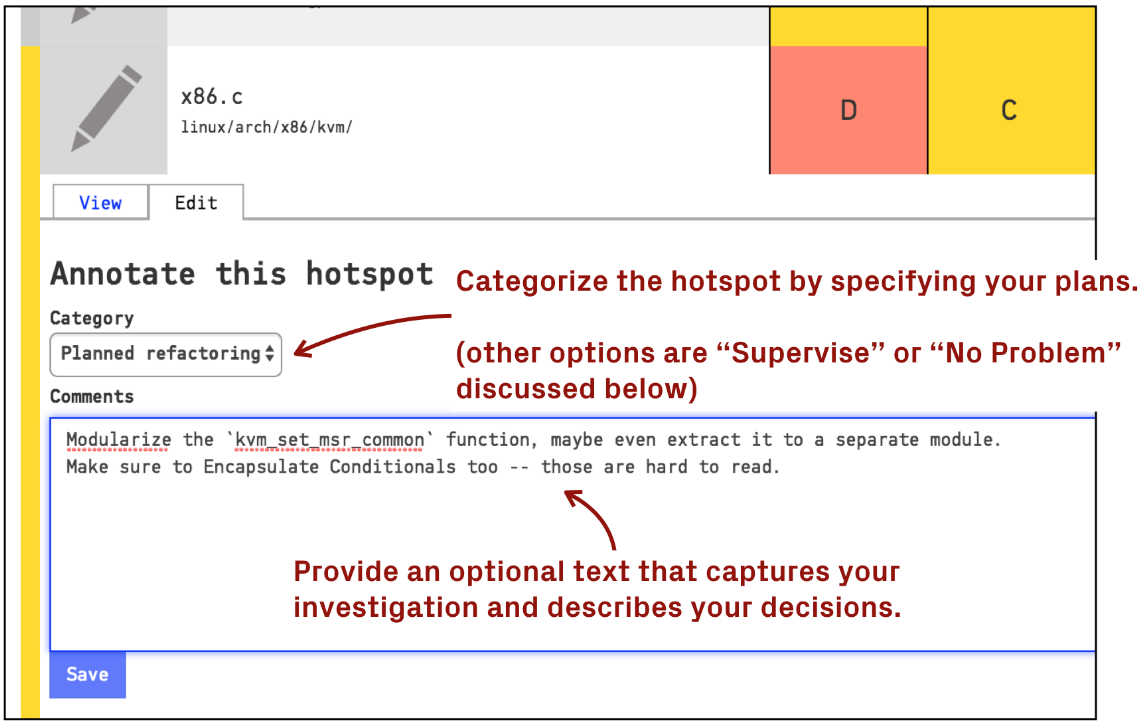
1. *Planned Refactoring*: Choose this category when you have investigated the code, perhaps with CodeScene's virtual code reviewer or its X-Ray analysis and see the need to pay-off some technical debt in the short term.
2. *Supervise*: Choose this category for code that might be acceptable for now, but that shouldn't grow worse.
3. *No Problem*: Choose this category to let CodeScene know that you don't consider the code a problem. Perhaps you plan a replacement of the hotspot code, or the code is good enough for your purposes.

You can specify the notes and categories in either the Code Biomarker view, Fig. 4.25 or in the Hotspots map through the virtual code reviewer, Fig. 4.28.

Once a note has been added, CodeScene will start to track it in every sub-sequent analysis. How CodeScene tracks a file dpends on the category you have assigned. Let's look at the differences between the categories in the next sections.

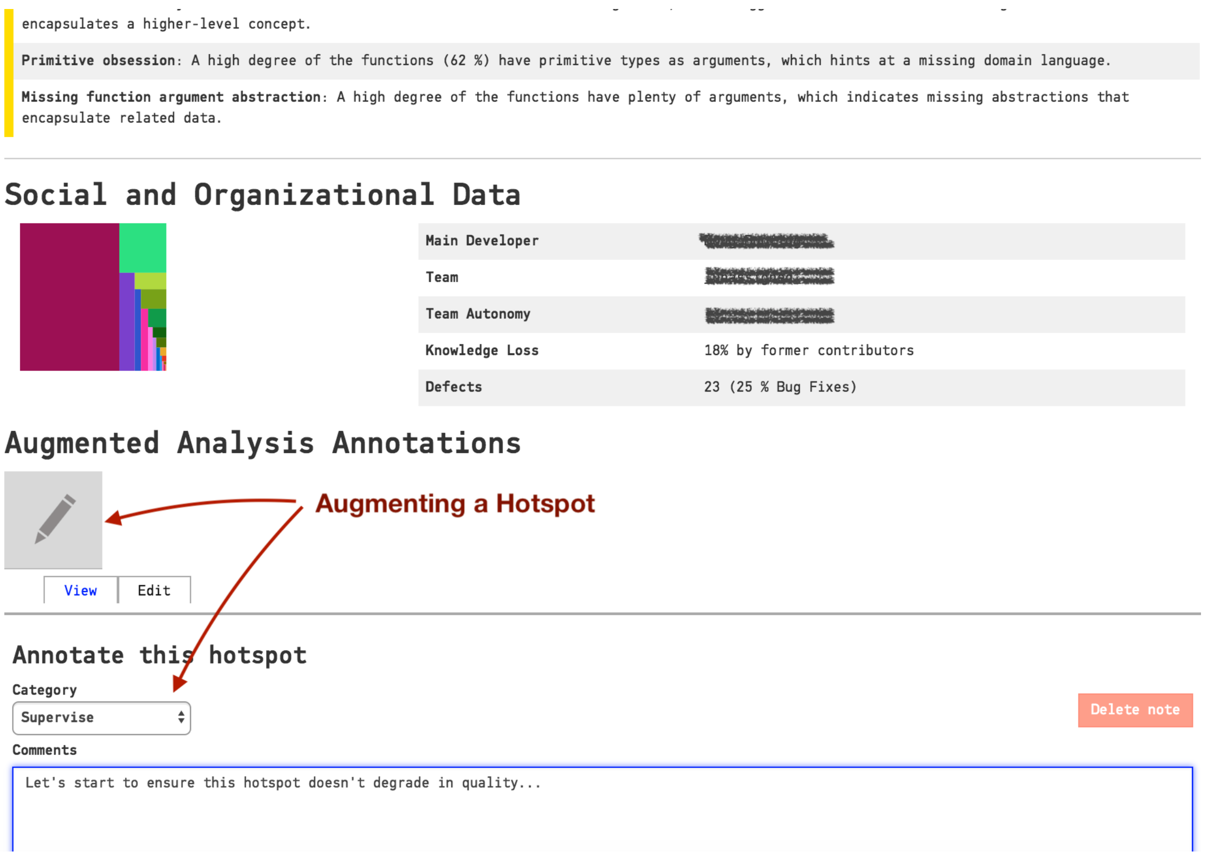
Analysis Of Planned Refactorings

When you specify a *Planned Refactoring*, CodeScene will do the following, depending on how the code evolves over time:


x86.c
linux/arch/x86/kvm/
D
C
View
Edit
Annotate this hotspot
Category
Planned refactoring
Comments
Save
Categorize the hotspot by specifying your plans.
(other options are "Supervise" or "No Problem" discussed below)
Provide an optional text that captures your investigation and describes your decisions.

```
Modularize the `kvm_set_msr_common` function, maybe even extract it to a separate module.
Make sure to Encapsulate Conditionals too -- those are hard to read.
```

Fig. 4.27: Add a note to any hotspot in the Code Biomarkers view.



encapsulates a higher-level concept.

Primitive obsession: A high degree of the functions (62 %) have primitive types as arguments, which hints at a missing domain language.

Missing function argument abstraction: A high degree of the functions have plenty of arguments, which indicates missing abstractions that encapsulate related data.

Social and Organizational Data

Main Developer	[redacted]
Team	[redacted]
Team Autonomy	[redacted]
Knowledge Loss	18% by former contributors
Defects	23 (25 % Bug Fixes)

Augmented Analysis Annotations

Let's start to ensure this hotspot doesn't degrade in quality...

Fig. 4.28: Notes can also be added to any file via the virtual code reviewer in CodeScene.

- *Code Degrades*: This will trigger a warning on the analysis dashboard since it violates the refactoring goal.
- *Code Improves*: This gives you a thumbs-up as the goal was met and the code is now in measureably better state.
- *No Change*: CodeScene understands that complex refactorings take time. However, if there's no clear improvement over the next months, CodeScene will warn you about it. Perhaps you want to re-consider your goal or prioritize the refactoring?

The decisions – improvement, degradation, or no change – are based on the evolution of the Code Biomarkers for that particular hotspot.

Fig. 4.29 shows an example of how this information is reflected in the Code Biomarkers.

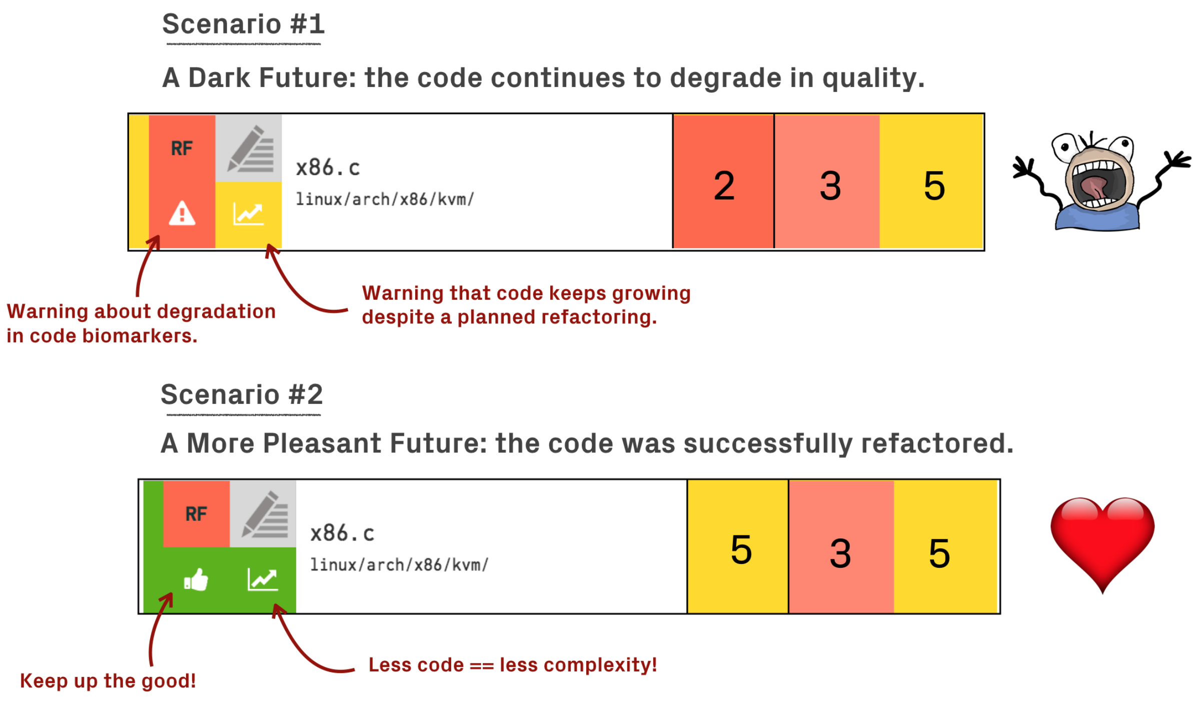

Scenario #1
A Dark Future: the code continues to degrade in quality.
RF
x86.c
linux/arch/x86/kvm/
2
3
5
Warning about degradation in code biomarkers.
Warning that code keeps growing despite a planned refactoring.
Scenario #2
A More Pleasant Future: the code was successfully refactored.
RF
x86.c
linux/arch/x86/kvm/
5
3
5
Keep up the good!
Less code == less complexity!

Fig. 4.29: An example of how refactoring progress is reported.

In case of the darker future, painted in Scenario 1, CodeScene will also draw your attention to it through an analysis warning as shown in Fig. 4.30 that indicates that the organization missed a goal.

In this case, the warning occurs because of a planned refactoring wasn't performed and the hotspot has continued to worsen. But the warnings are also fired in other situations. Let's look at hotspot supervision to see another example.

Analysis Of Supervised Hotspots

Like we discussed earlier, you might chose a different trade-off and wait before undertaking a larger refactoring. Perhaps you know that the code in question is feature complete, or maybe you conclude that the code is acceptable after all. In that case you want to ensure that it stays that way. You do that by delegating the responsibility of supervising such hotspots to CodeScene as shown in Fig. 4.31.

With hotspot supervision, you ensure that no new technical debt is taken on without you getting an alert from CodeScene.

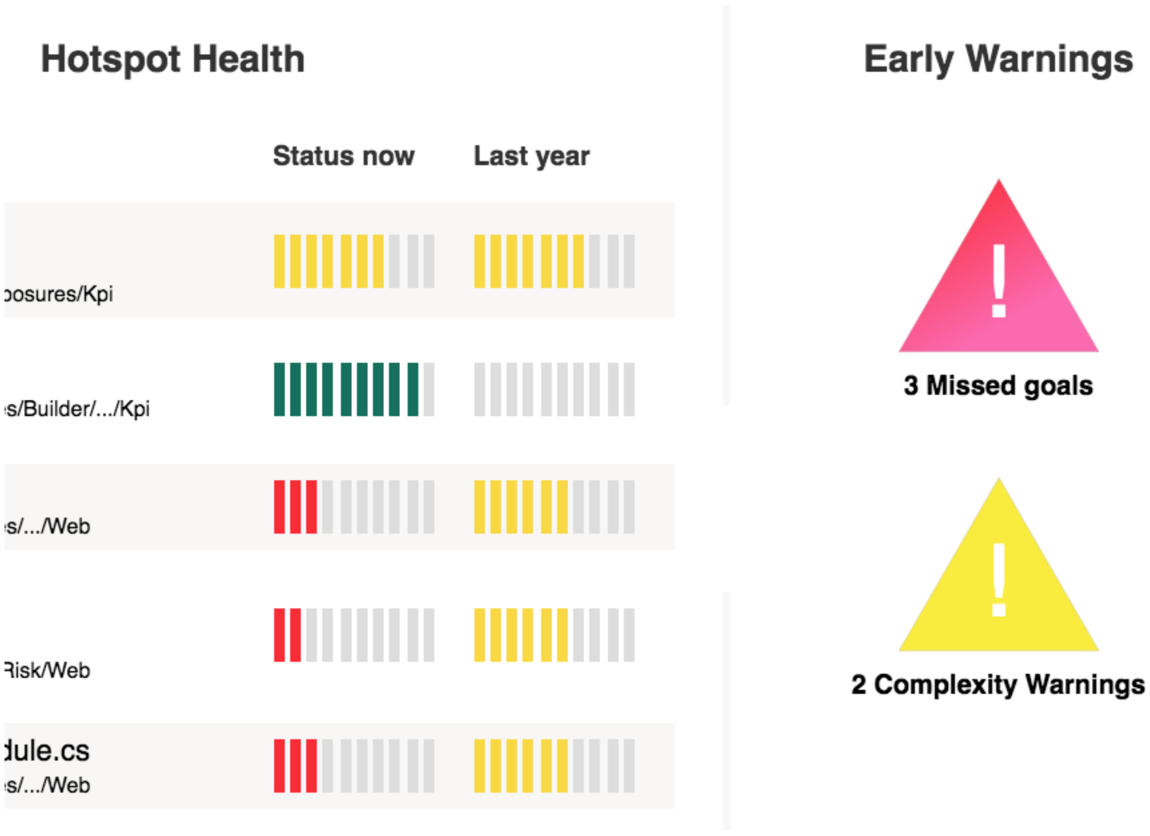


Fig. 4.30: A warning that a planned refactoring hasn't been performed.

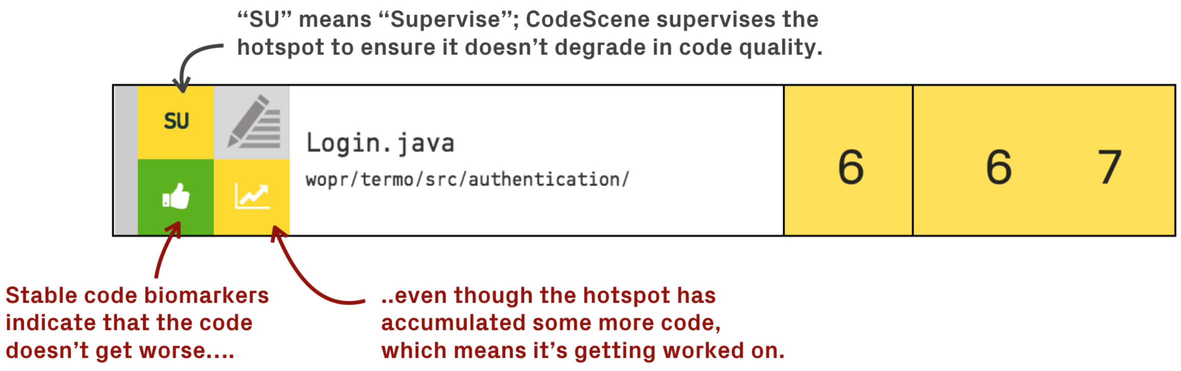

"SU" means "Supervise"; CodeScene supervises the hotspot to ensure it doesn't degrade in code quality.
SU
Login.java
wopr/termo/src/authentication/
6
6
7
Stable code biomarkers indicate that the code doesn't get worse....
..even though the hotspot has accumulated some more code, which means it's getting worked on.

Fig. 4.31: CodeScene acts as an extra team member that constantly supervises code at risk to grow worse.

Analysis Of Code Marked As "No Problem"

False positives show up every now and then in all automated code analyses and CodeScene is no exception. For example, there might be this gigantic hotspot file that is worked on all the time but just contains common declarations like enumerations or constants. While that might not be an award-winning design, such code is usually not a primary driver of excess development costs. Or maybe you do identify a problematic hotspot but know – again: contextual information – that this code will be replaced by a new library next month.

In that case you can make the decision to ask CodeScene to ignore a specific hotspot, and it won't show up as prioritized technical debt in your next analysis.

However, there's always a danger of suppressing warning signs or problems. Hence, CodeScene keeps analyzing ignored hotspots in the background even if you explicitly marked them as *No Problem*. Should something dramatic happen to that code, CodeScene will point your attention to it as shown in Fig. 4.32.

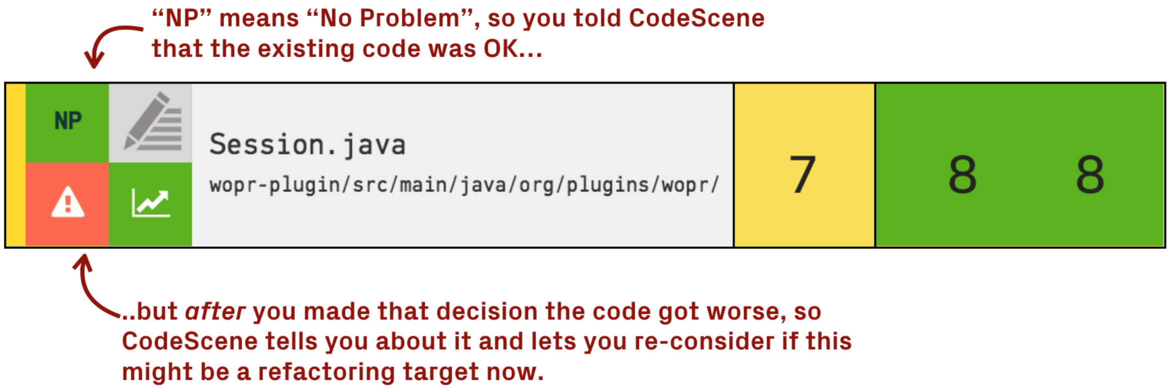
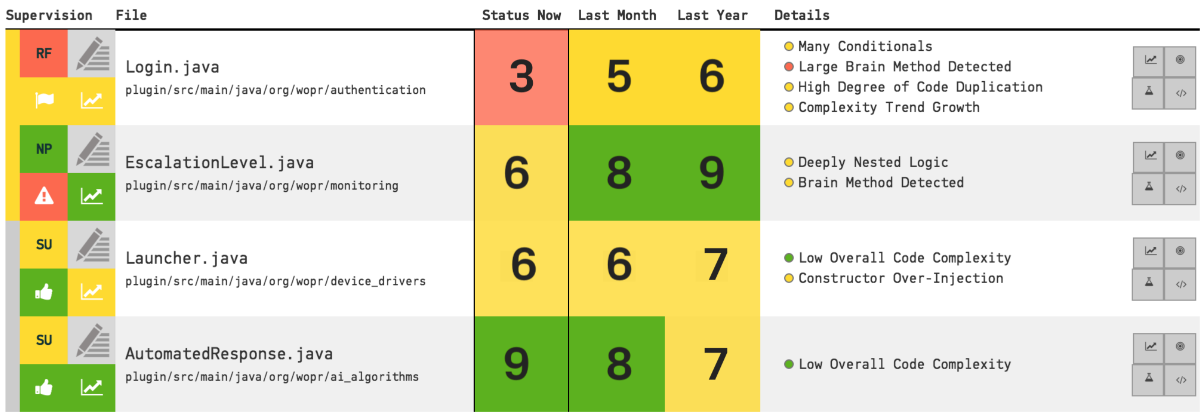

"NP" means "No Problem", so you told CodeScene that the existing code was OK...
NP
Session.java
wopr-plugin/src/main/java/org/plugins/wopr/
7
8
8
..but *after* you made that decision the code got worse, so CodeScene tells you about it and lets you re-consider if this might be a refactoring target now.

Fig. 4.32: In the background, CodeScene continues to scan hotspots marked as No Problem to ensure they stay that way.

CodeScene will also issue a warning on the project's dashboard so that you can re-consider the goals for that hotspot.

View Your Technical Debt At A Glance

By augmenting your hotspots with contextual information and refactoring plans, you can then view your prioritized hotspots in the context of your goals and situation as shown in Fig. 4.33.



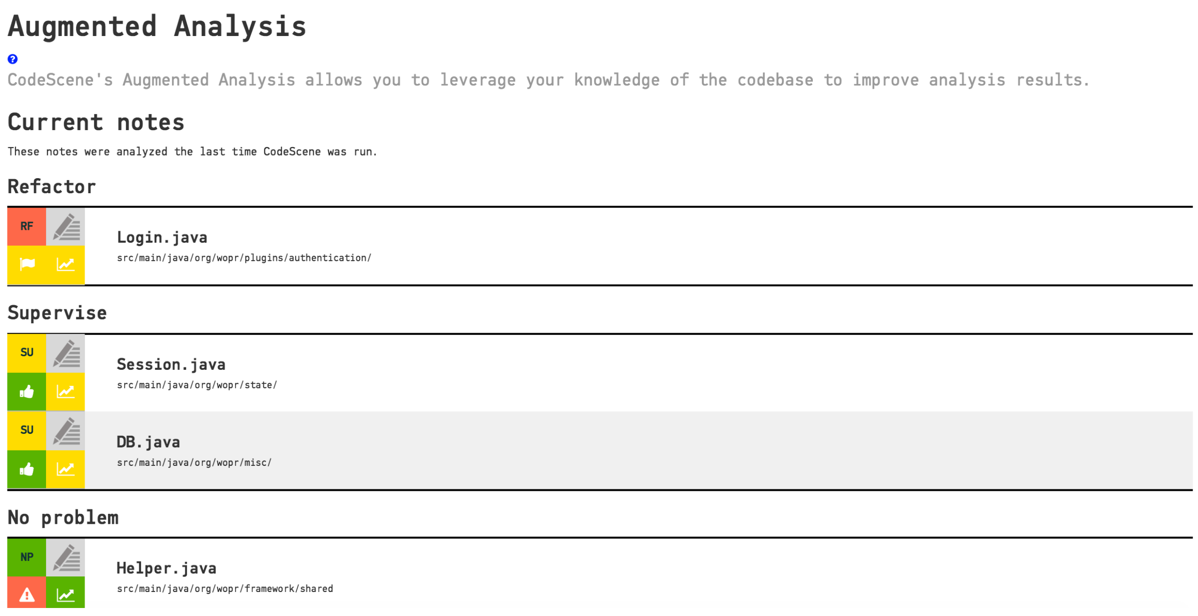
Supervision	File	Status Now	Last Month	Last Year	Details
RF	Login.java plugin/src/main/java/org/wopr/authentication	3	5	6	● Many Conditionals ● Large Brain Method Detected ● High Degree of Code Duplication ● Complexity Trend Growth
NP	EscalationLevel.java plugin/src/main/java/org/wopr/monitoring	6	8	9	● Deeply Nested Logic ● Brain Method Detected
SU	Launcher.java plugin/src/main/java/org/wopr/device_drivers	6	6	7	● Low Overall Code Complexity ● Constructor Over-Injection
SU	AutomatedResponse.java plugin/src/main/java/org/wopr/ai_algorithms	9	8	7	● Low Overall Code Complexity

Fig. 4.33: View your hotspots in context of your decisions and plans.

Using CodeScene's dashboard, you can even have the data updated in real-time as the code is being developed. That way, the whole organization gets to share a common view of what the state of the code actually looks like and how it evolves. That's the first step towards true improvements.

Manage Your Intelligent Notes From The Dashboard

Most of the time you will interact with the intelligent notes via the virtual code reviewer or the Code Biomarkers view. But you can also get an overview of all notes and administrate them on a separate dashboard as shown in Fig. 4.34.



Augmented Analysis

CodeScene's Augmented Analysis allows you to leverage your knowledge of the codebase to improve analysis results.

Current notes

These notes were analyzed the last time CodeScene was run.

Refactor

RF Login.java src/main/java/org/wopr/plugins/authentication/

Supervise

SU Session.java src/main/java/org/wopr/state/

SU DB.java src/main/java/org/wopr/misc/

No problem

NP Helper.java src/main/java/org/wopr/framework/shared

Fig. 4.34: Manage the intelligent notes from their dashboard.

The dashboard is particularly interesting for the hotspots classified as *No Problem* as they won't show up in the other analyses unless CodeScene found a growing problem in them.

Know The Edge Cases When Tracking Hotspots

CodeScene does its best to track the notes you have attached to hotspots even if you move or rename the hotspot files. However, in some situations this isn't possible. The reasons are due to the way Git works.

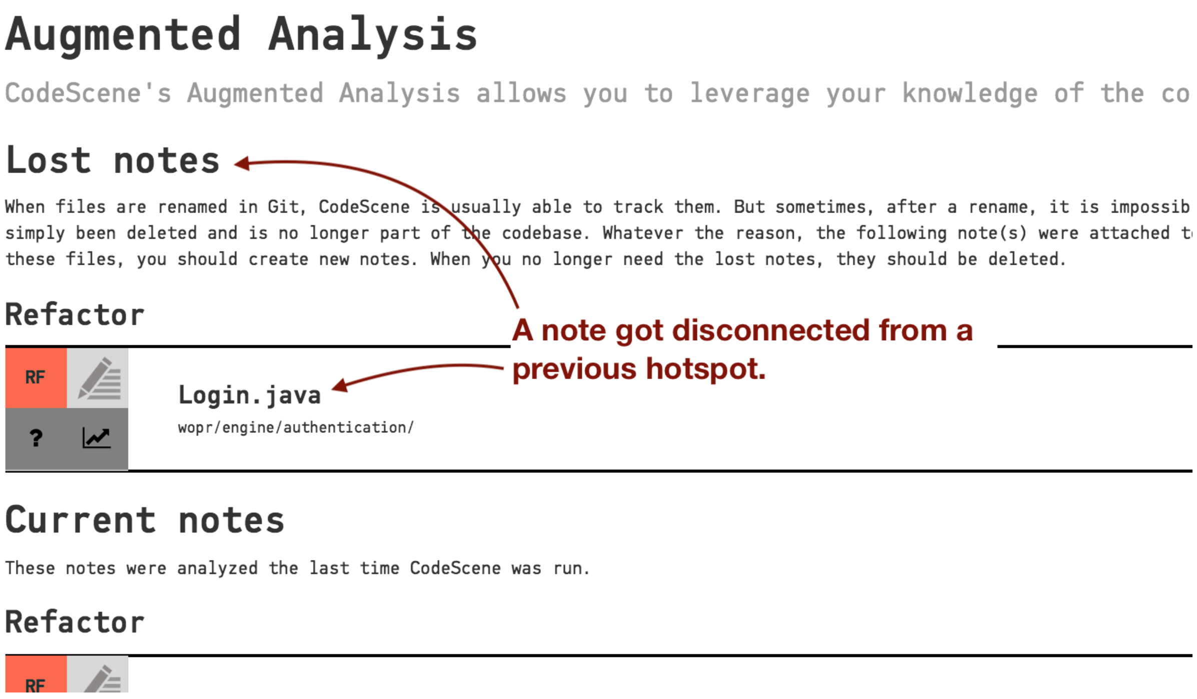
When CodeScene cannot find the file referenced by a note, that note will appear in the list of "Lost Notes" in the Augmented Analysis Dashboard shown in Fig. 4.35.

The interface for lost notes lets you to decide what to do:

- A file has been removed from the project. If this is the case, it's time to remove the note as well. Just click on the Pen icon and select "Delete".
- A file (or an ancestor directory) has been renamed or moved. You should create a new note through the usual procedure, then delete the lost note.

The following situations are known to cause a *lost note*:

- Deleted content: The most common reason for a lost note is that the original hotspot has been deleted.
- Two-step renaming of content: Normal file renames work fine, but if the deletion step and the adding of the new file are performed in separate commits then CodeScene won't be able to maintain the link from note to content.


Augmented Analysis

CodeScene's Augmented Analysis allows you to leverage your knowledge of the co

Lost notes

When files are renamed in Git, CodeScene is usually able to track them. But sometimes, after a rename, it is impossib
simply been deleted and is no longer part of the codebase. Whatever the reason, the following note(s) were attached t
these files, you should create new notes. When you no longer need the lost notes, they should be deleted.

Refactor

Login.java
wopr/engine/authentication/

Current notes

These notes were analyzed the last time CodeScene was run.

Refactor

Fig. 4.35: Manage disconnected notes from the dashboard.

4.2.4 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. 4.36.

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. 4.37.

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. 4.37 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:

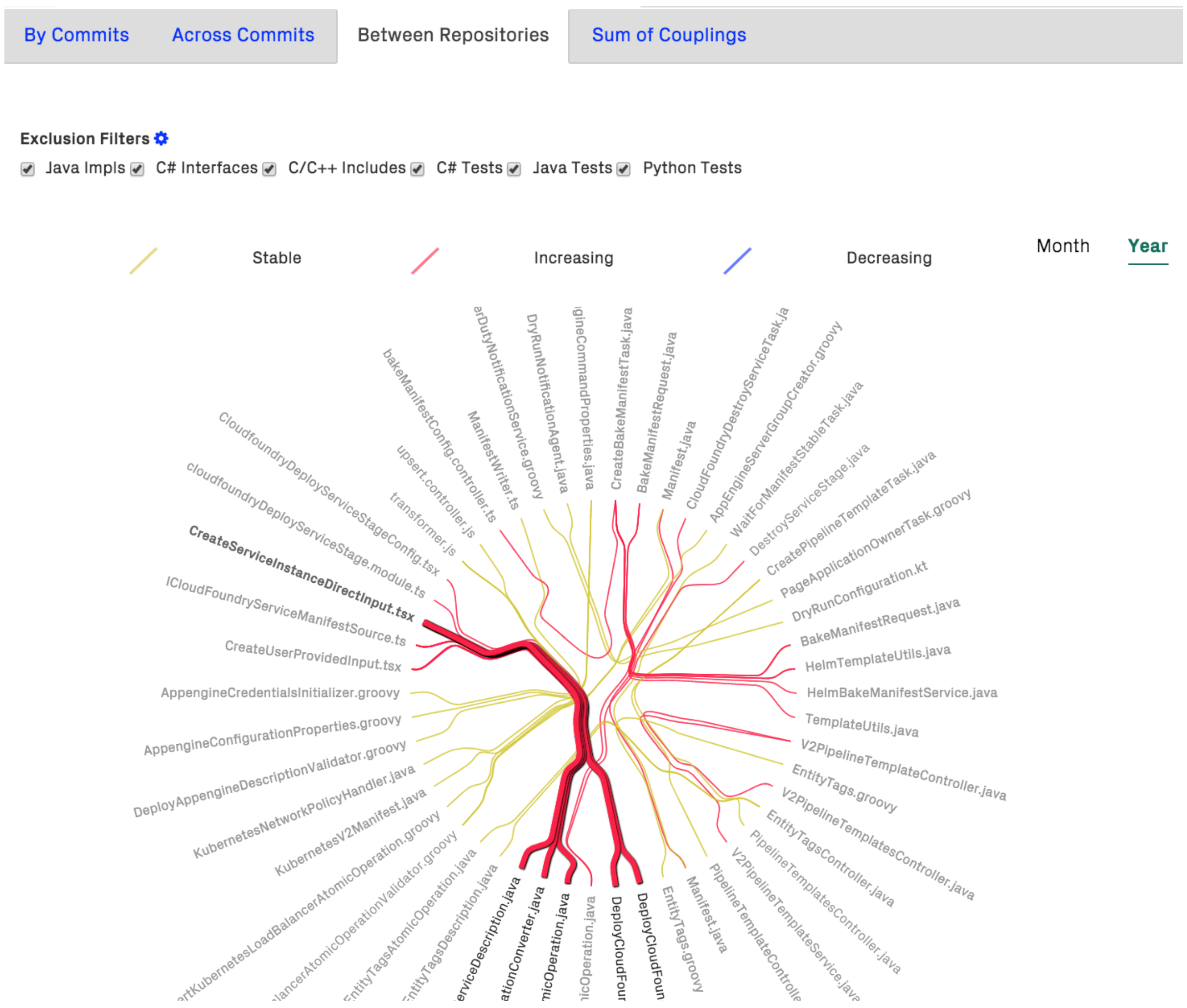
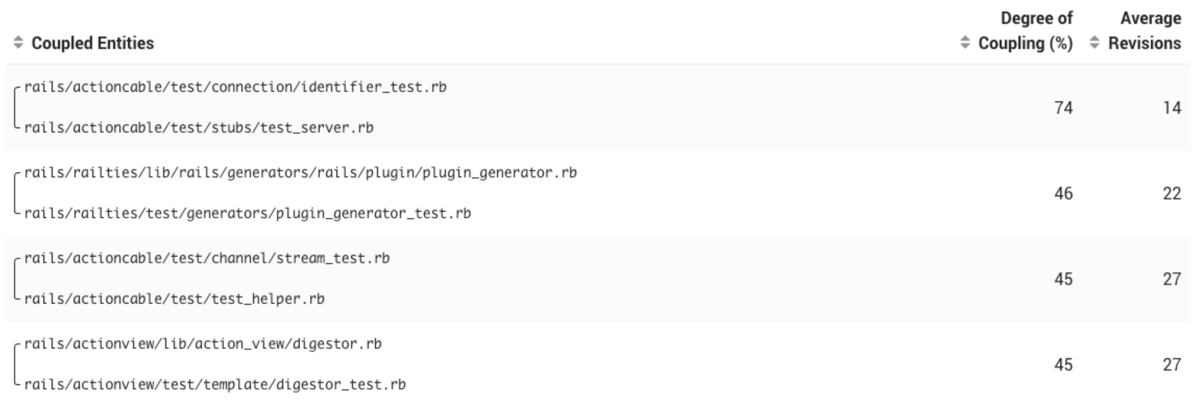

By Commits
Across Commits
Between Repositories
Sum of Couplings
Exclusion Filters
☑ Java Impls ☑ C# Interfaces ☑ C/C++ Includes ☑ C# Tests ☑ Java Tests ☑ Python Tests
Stable
Increasing
Decreasing
Month
Year
CreateServiceInstanceDirectInput.tsx
HelmTemplateUtils.java
HelmBakeManifestService.java
TemplateUtils.java
V2PipelineTemplateController.java
EntityTags.groovy
AppengineCredentialsInitializer.groovy
AppengineConfigurationProperties.groovy
DeployAppengineDescriptionValidator.groovy
KubernetesNetworkPolicyHandler.java
KubernetesV2Manifest.groovy
CreateUserProvidedInput.tsx
ICloudFoundryServiceManifestSource.ts
cloudfoundryDeployServiceStage.module.ts
CloudfoundryDeployServiceStageConfig.tsx
bakeManifestConfig.controller.ts
ManifestWriter.ts
DryRunNotificationAgent.java
CreateBakeManifestTask.java
BakeManifestRequest.java
Manifest.java
CloudFoundryDestroyServiceTask.java
AppEngineServerGroupCreator.groovy
WaitForManifestStableTask.java
DestroyServiceStage.java
CreatePipelineTemplateTask.java
PageApplicationOwnerTask.groovy
DryRunConfiguration.kt
BakeManifestRequest.java
DeployCloudFoundry
EntityTags.groovy
Manifest.java
V2PipelineTemplatesController.java
EntityTagsController.java
PipelineTemplatesController.java
V2PipelineTemplatesService.java

Fig. 4.36: 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. 4.37: The temporal coupling table gives you all the details.

- Detect software clones (aka copy-paste code).
- Evaluate the relevance your unit tests.
- Detect architectural decay.
- 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. 4.38 shows an example of such a case.

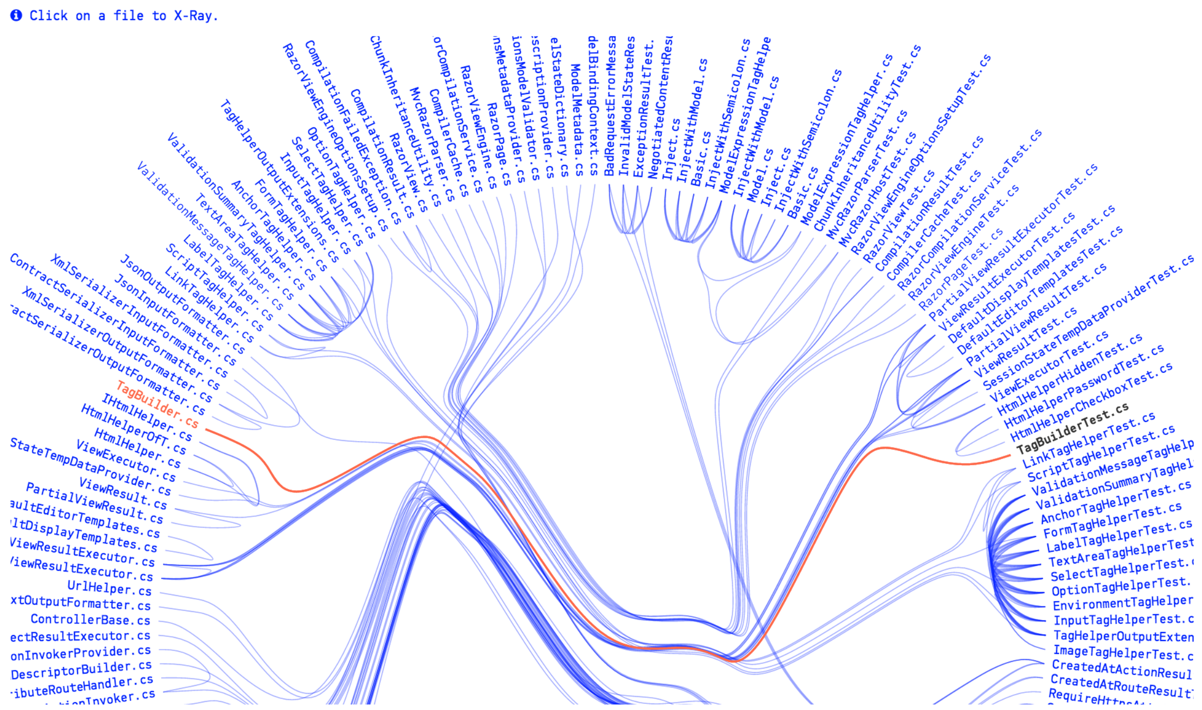


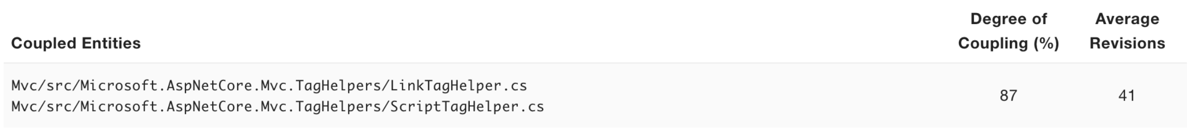
Fig. 4.38: Temporal sample on unit test.

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. 4.39 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. 4.39: Unexpected temporal coupling.

The table in Fig. 4.39 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 71)).

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.

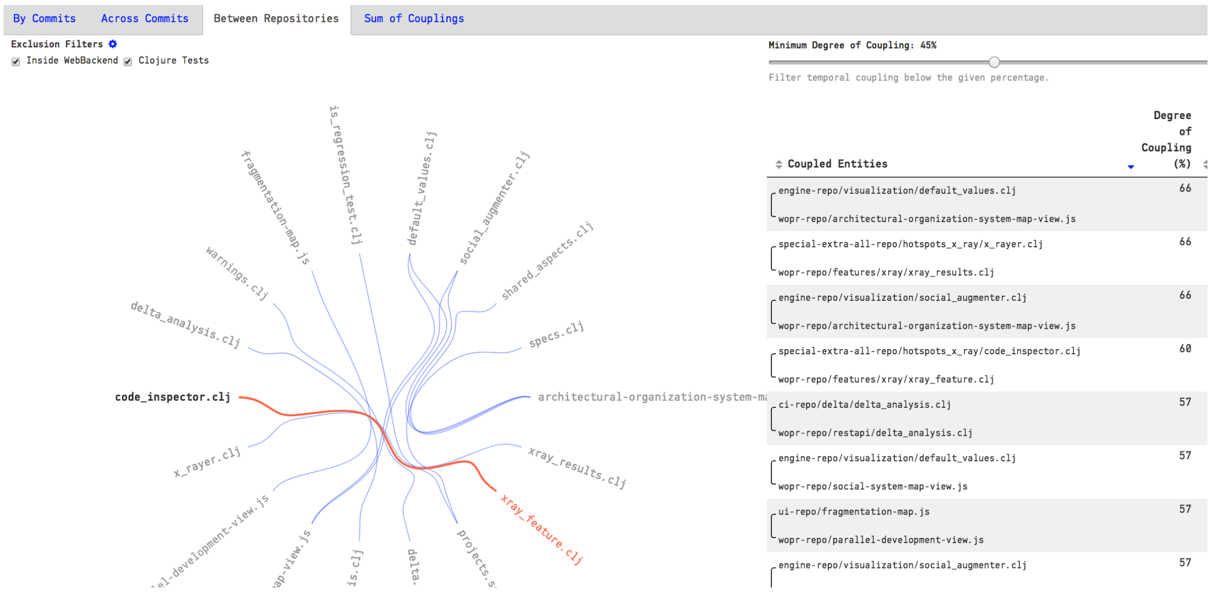
Investigate Temporal Dependencies across Architectural Boundaries

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.

Detect Change Patterns Across Repositories

CodeScene's temporal coupling filters can be used to make it easier to detect changes that ripple across repository boundaries. However, if you have tens or hundreds of repositories it's going to be painful to configure. To solve that CodeScene provides a special view that only focuses on the temporal couplings that cross repository boundaries, as show in Fig. 4.40.



✓ Inside WebBackend ✓ Clojure Tests

Minimum Degree of Coupling: 45%

Coupled Entities	Degree of Coupling (%)
engine-repo/visualization/default_values.clj wopr-repo/architectural-organization-system-map-view.js	66
special-extra-all-repo/hotspots_x_ray/x_rayer.clj wopr-repo/features/xray/xray_results.clj	66
engine-repo/visualization/social_augmenter.clj wopr-repo/architectural-organization-system-map-view.js	66
special-extra-all-repo/hotspots_x_ray/code_inspector.clj wopr-repo/features/xray/xray_feature.clj	60
ci-repo/delta/delta_analysis.clj wopr-repo/restapi/delta_analysis.clj	57
engine-repo/visualization/default_values.clj wopr-repo/social-system-map-view.js	57
ui-repo/fragmentation-map.js wopr-repo/parallel-development-view.js	57
engine-repo/visualization/social_augmenter.clj	57

Fig. 4.40: Detect temporal coupling between repositories.

This view provides detailed information on exactly what files in different repositories that have implicit dependencies between them. Again, look for surprising patterns that violate your expectations or architectural principles.

Once you've identified such change patterns you use X-Ray to resolve the coupling on a function level. Since X-Ray works across repository boundaries, you're usually able to uncover surprising patterns that aren't visible in the code nor the Git repository (see *X-Ray* (page 71) for more details).

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. 4.41 shows an example from Roslyn, Microsoft's open source compiler platform.

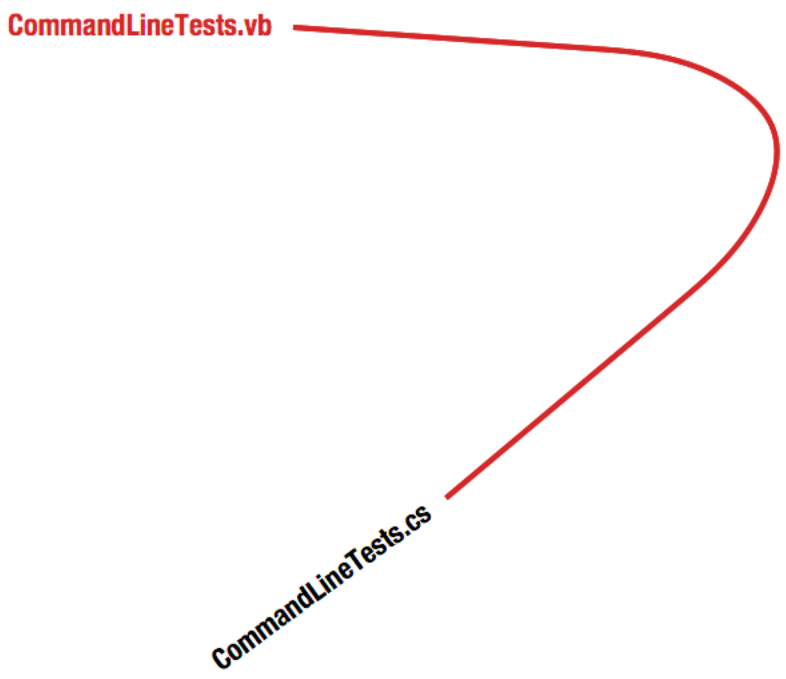


Fig. 4.41: Temporal coupling between languages.

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.

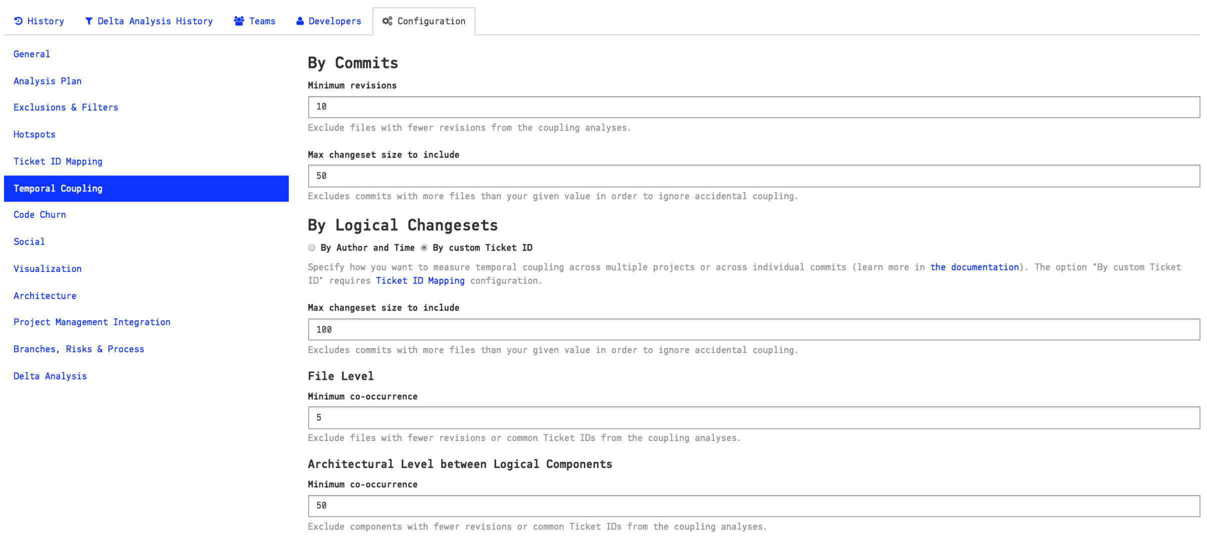
Dig Deeper with Sum of Coupling

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. 4.42.



By Commits

Minimum revisions: 10

Exclude files with fewer revisions from the coupling analyses.

Max changeset size to include: 50

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

By Logical Changesets

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.

Max changeset size to include: 100

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

File Level

Minimum co-occurrence: 5

Exclude files with fewer revisions or common Ticket IDs from the coupling analyses.

Architectural Level between Logical Components

Minimum co-occurrence: 50

Exclude components with fewer revisions or common Ticket IDs from the coupling analyses.

Fig. 4.42: Temporal coupling configuration.

The temporal coupling thresholds are configured on two different levels that correlate to the resulting analysis views:

1. *By Commits*: This configuration refers to files that tend to be changed as part of the same commit. This is the strongest level of coupling, and the primary inspection point.
2. *By Logical Changesets*: This configuration refers to files and logical components that might be changed in _separate_ commits, maybe even located in different repositories. CodeScene lets you specify the threshold separately as you typically want to use lower thresholds when identifying patterns across commits in different repositories as it's an aggregated metric.

As a specific example configuration, the settings in Fig. 4.42 for the *By Commits* section means that the temporal coupling algorithm will respect the following thresholds:

1. Ignore all files with less than 10 commits since the coupling trend isn't strong enough yet.
2. 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 the initial analyses. You typically lower the thresholds in case you don't find any temporal coupling.

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.

4.2.5 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 these 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. 4.43.

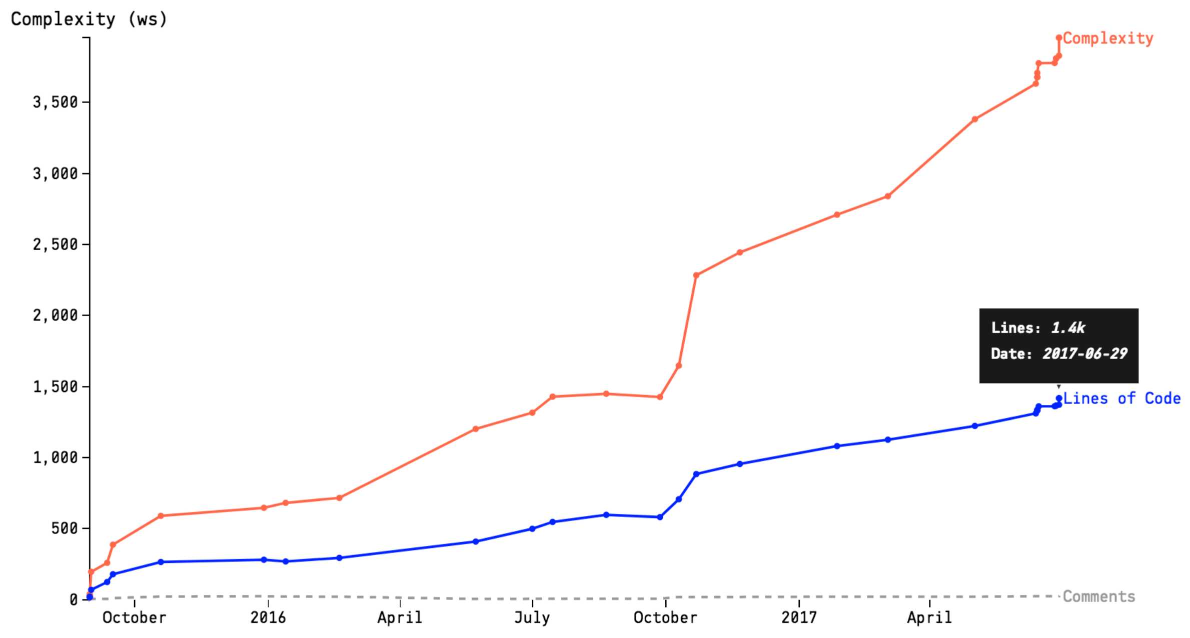


Fig. 4.43: A complexity trend sample.

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 evidenced by the Complexity/Lines of Code ratio shown in Fig. 4.44, the complexity grows non-linearly to the amount of new code, which indicates that the code in the hotspot is getting 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. 4.45 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.

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:

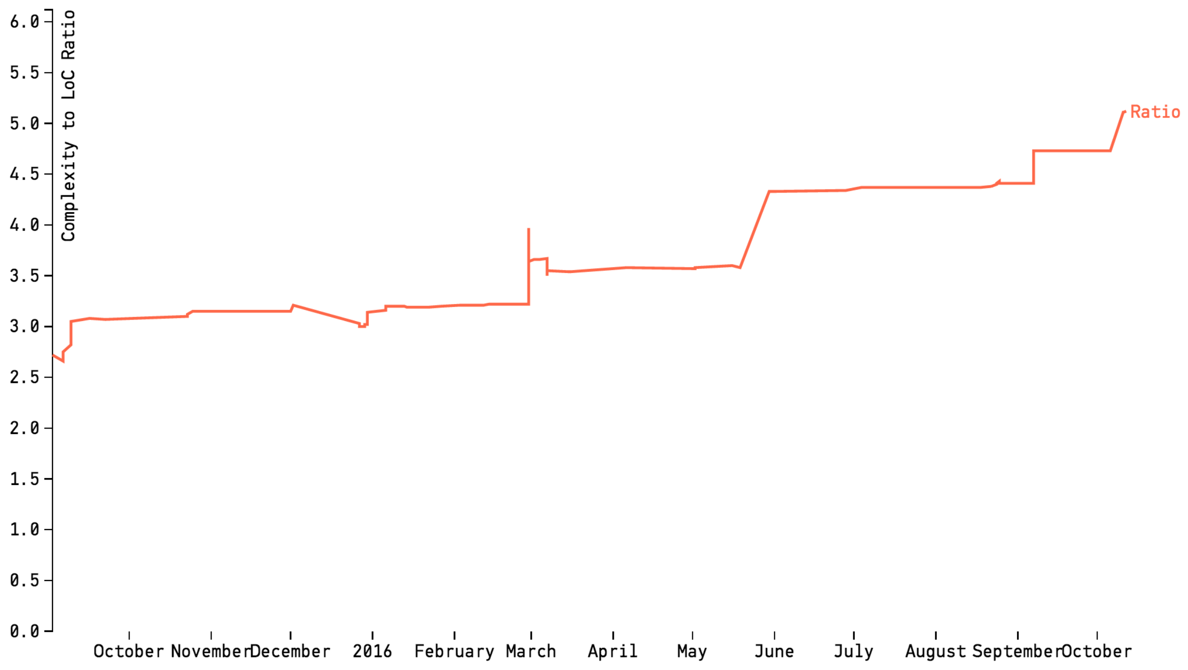


Fig. 4.44: The ration between complexity and lines of code accumulation.

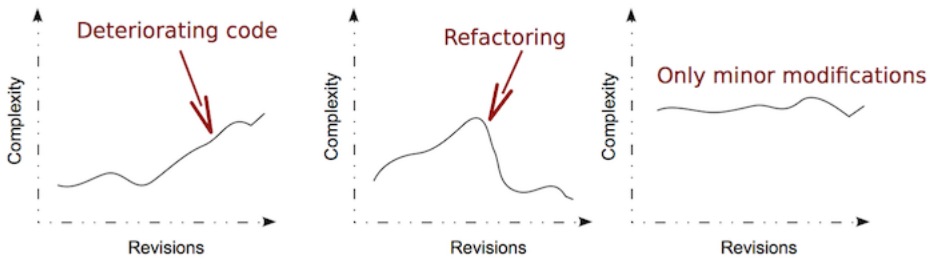


Fig. 4.45: Complexity trend patterns you might find in a codebase.

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, but 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 becomes a hotspot 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. Do you see the second pattern, "Refactoring"? A downward slope in a complexity trend is a good sign. It either means that your code is getting 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. 4.46 illustrates one such scary case.

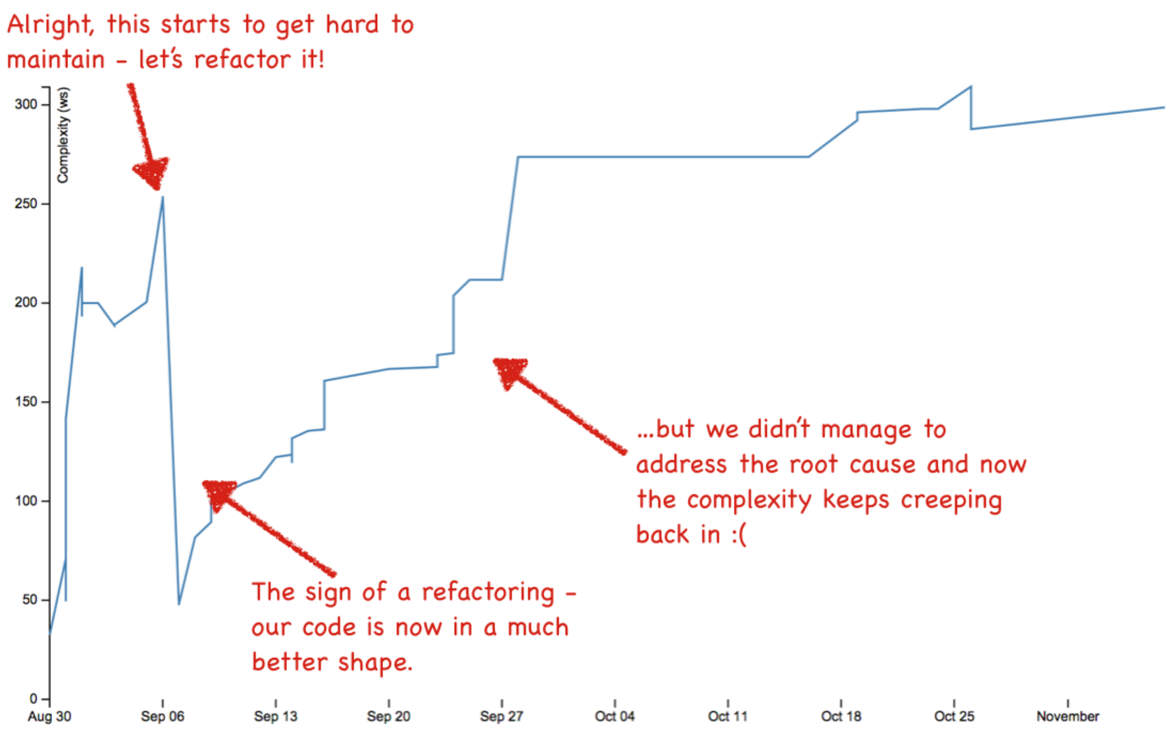
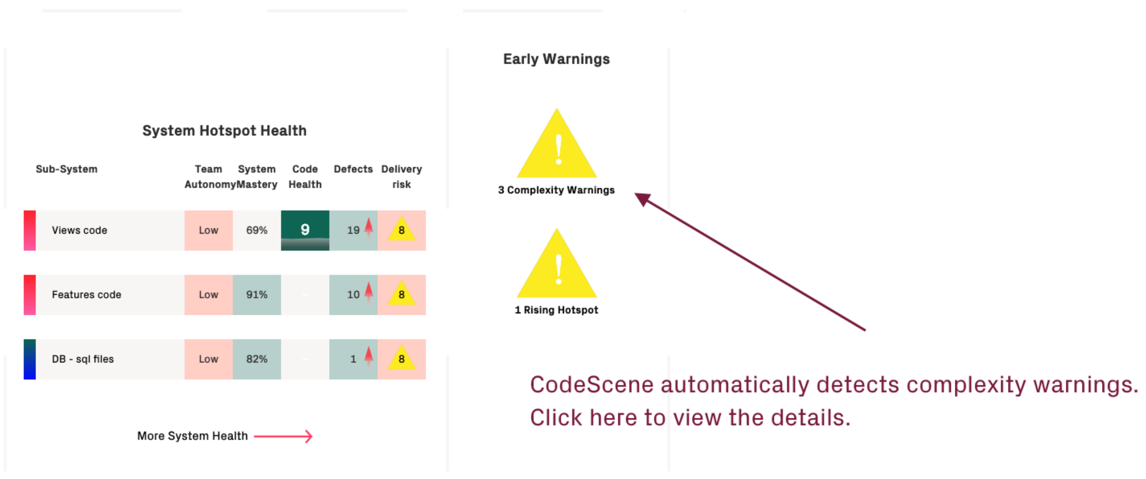


Fig. 4.46: Example on failed refactoring.

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; CodeScene will even do it automatically for you, as illustrated in Fig. 4.47. Those complexity trend warnings are triggered when the code complexity in any part of your code starts to grow at a rapid rate.



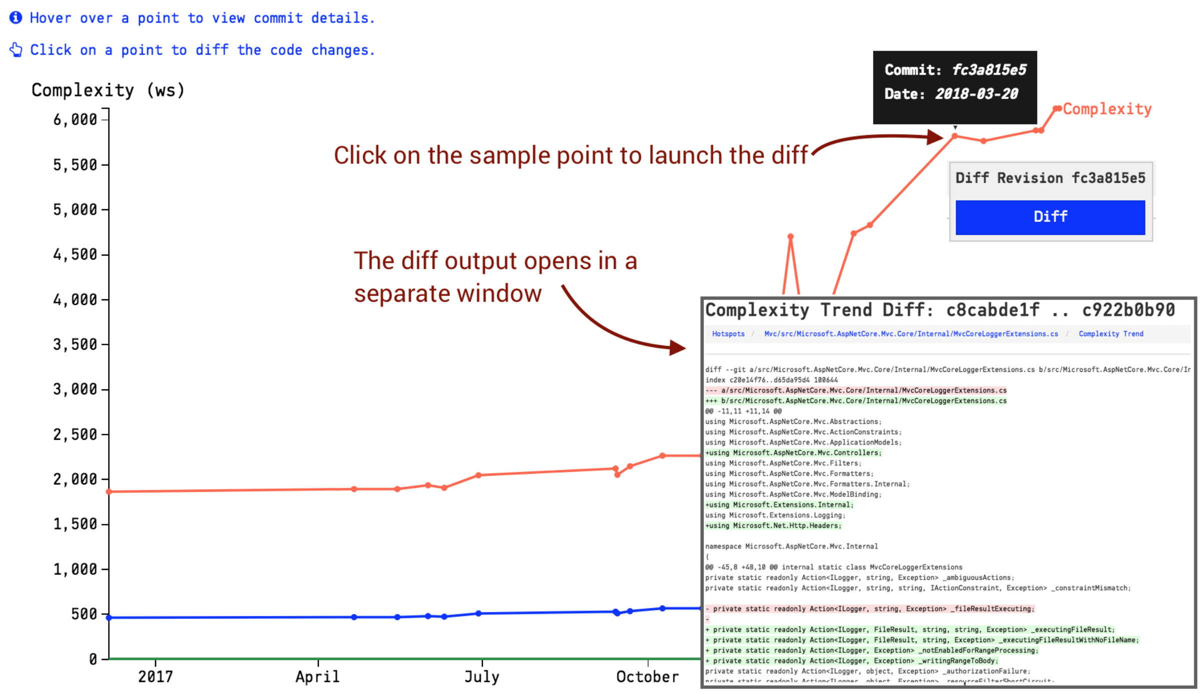
Sub-System	Team Autonomy	System Mastery	Code Health	Defects	Delivery risk
Views code	Low	69%	9	19	8
Features code	Low	91%		10	8
DB - sql files	Low	82%		1	8

Fig. 4.47: CodeScene supervises your complexity trends.

Diff your Complexity Trends

CodeScene provides an automated *diff* between the sample points in a complexity trend. A diff is a useful investigative tool when you look to explain why a trend suddenly increased or dropped in complexity.

You run a *diff* by clicking on one of the sample points in the complexity trend, as shown in Fig. 4.48.


Hover over a point to view commit details.
Click on a point to diff the code changes.
Complexity (ws)
6,000
5,500
5,000
4,500
4,000
3,500
3,000
2,500
2,000
1,500
1,000
500
0
2017
April
July
October
Commit: fc3a815e5
Date: 2018-03-20
Complexity
Click on the sample point to launch the diff
Diff Revision fc3a815e5
Diff
The diff output opens in a separate window
Complexity Trend Diff: c8cabde1f .. c922b0b90
Hotspots / Mvc/src/Microsoft.AspNetCore.Mvc.Core/Internal/MvcCoreLoggerExtensions.cs / Complexity Trend

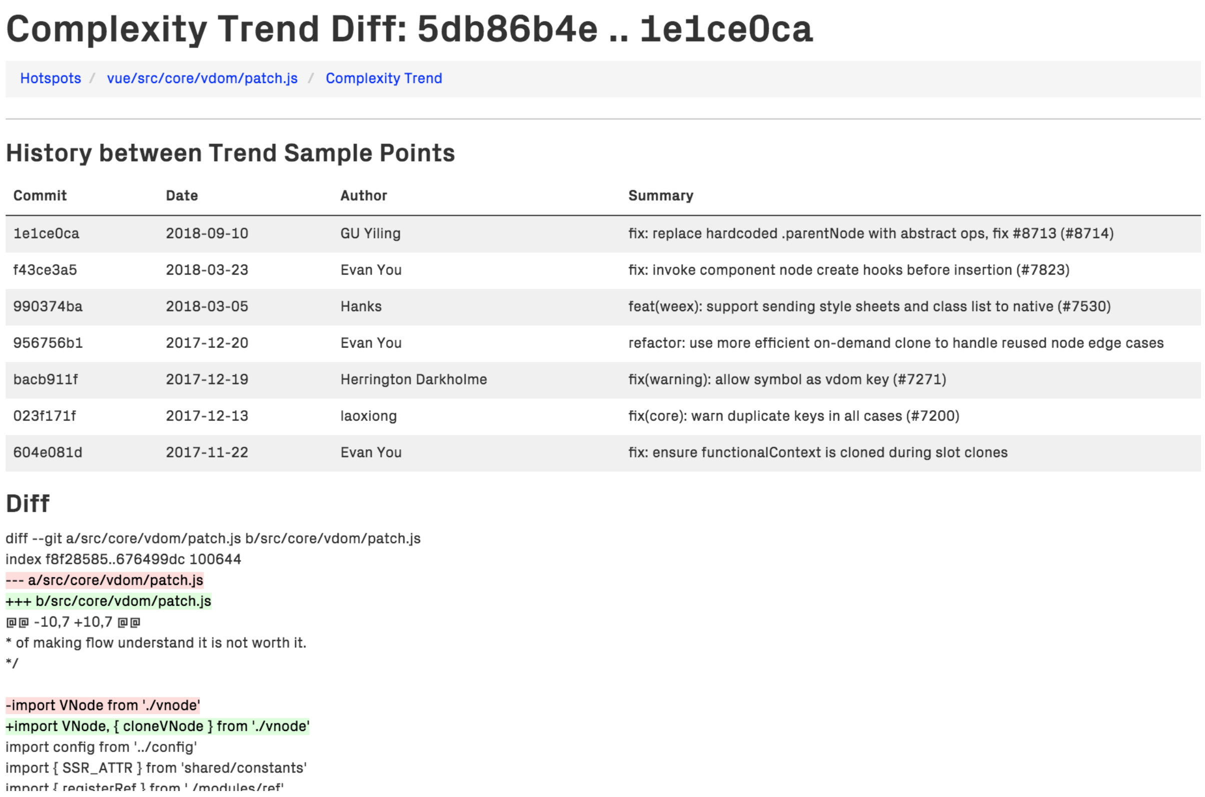
```
diff --git a/src/Microsoft.AspNetCore.Mvc.Core/Internal/MvcCoreLoggerExtensions.cs b/src/Microsoft.AspNetCore.Mvc.Core/Ir
index c20e14f76..d65da95d4 100644
--- a/src/Microsoft.AspNetCore.Mvc.Core/Internal/MvcCoreLoggerExtensions.cs
+++ b/src/Microsoft.AspNetCore.Mvc.Core/Internal/MvcCoreLoggerExtensions.cs
@@ -11,11 +11,14 @@
using Microsoft.AspNetCore.Mvc.Abstractions;
using Microsoft.AspNetCore.Mvc.ActionConstraints;
using Microsoft.AspNetCore.Mvc.ApplicationModels;
+using Microsoft.AspNetCore.Mvc.Controllers;
using Microsoft.AspNetCore.Mvc.Filters;
using Microsoft.AspNetCore.Mvc.Formatters;
using Microsoft.AspNetCore.Mvc.Formatters.Internal;
using Microsoft.AspNetCore.Mvc.ModelBinding;
+using Microsoft.Extensions.Internal;
using Microsoft.Extensions.Logging;
+using Microsoft.Net.Http.Headers;

namespace Microsoft.AspNetCore.Mvc.Internal
{
@@ -45,8 +48,10 @@ internal static class MvcCoreLoggerExtensions
private static readonly Action<ILogger, string, Exception> _ambiguousActions;
private static readonly Action<ILogger, string, string, IActionConstraint, Exception> _constraintMismatch;

- private static readonly Action<ILogger, string, Exception> _fileResultExecuting;
-
+ private static readonly Action<ILogger, FileResult, string, string, Exception> _executingFileResult;
+ private static readonly Action<ILogger, FileResult, string, Exception> _executingFileResultWithNoFileName;
+ private static readonly Action<ILogger, Exception> _notEnabledForRangeProcessing;
+ private static readonly Action<ILogger, Exception> _writingRangeToBody;
private static readonly Action<ILogger, object, Exception> _authorizationFailure;
private static readonly Action<ILogger, object, Exception> _resourceFilterShortCircuit;
```

Fig. 4.48: Diff a complexity trend by clicking on a sample point.

The *diff* is done between the sample points in the complexity trend graph, which might include several commits. Hence, the diff view includes the history between those samples points as shown in Fig. 4.49.



Complexity Trend Diff: 5db86b4e .. 1e1ce0ca

Hotspots / vue/src/core/vdom/patch.js / Complexity Trend

History between Trend Sample Points

Commit	Date	Author	Summary
1e1ce0ca	2018-09-10	GU Yiling	fix: replace hardcoded .parentNode with abstract ops, fix #8713 (#8714)
f43ce3a5	2018-03-23	Evan You	fix: invoke component node create hooks before insertion (#7823)
990374ba	2018-03-05	Hanks	feat(weex): support sending style sheets and class list to native (#7530)
956756b1	2017-12-20	Evan You	refactor: use more efficient on-demand clone to handle reused node edge cases
bacb911f	2017-12-19	Herrington Darkholme	fix(warning): allow symbol as vdom key (#7271)
023f171f	2017-12-13	laoxiong	fix(core): warn duplicate keys in all cases (#7200)
604e081d	2017-11-22	Evan You	fix: ensure functionalContext is cloned during slot clones

Diff

```
diff --git a/src/core/vdom/patch.js b/src/core/vdom/patch.js
index f8f28585..676499dc 100644
--- a/src/core/vdom/patch.js
+++ b/src/core/vdom/patch.js
@@ -10,7 +10,7 @@
* of making flow understand it is not worth it.
*/

-import VNode from './vnode'
+import VNode, { cloneVNode } from './vnode'
import config from '../config'
import { SSR_ATTR } from 'shared/constants'
```

Fig. 4.49: The complexity trend diff includes the history between the samples points.

Limitation: Please note that in the current version, CodeScene won't be able to *diff* files that have been renamed or moved. This functionality is likely to be added in a future version.

What's "Complexity" anyway?

All right, 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. 4.50

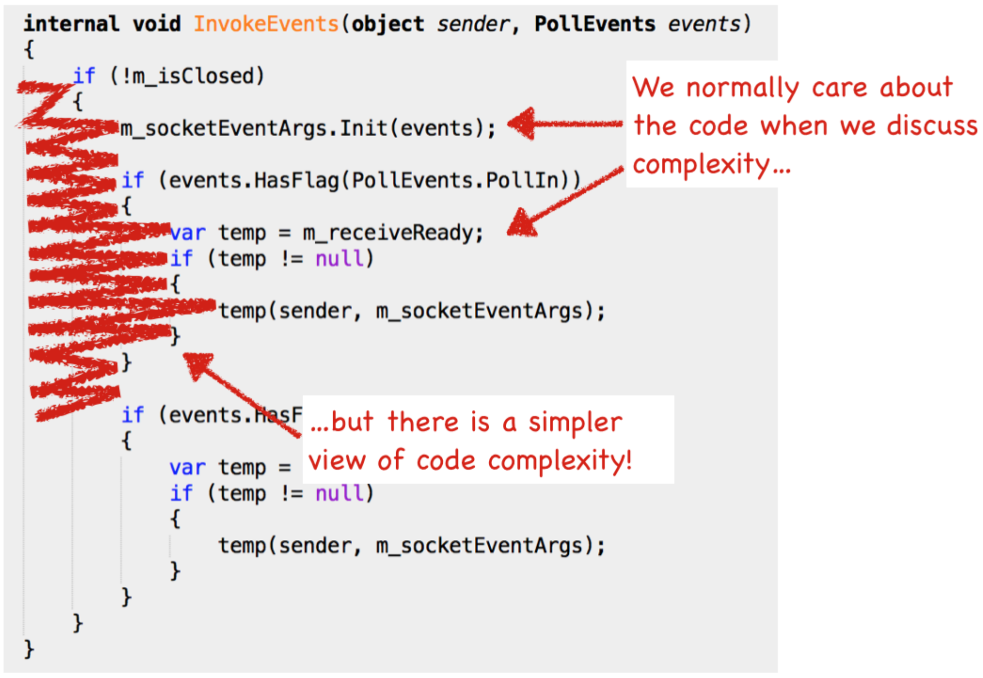
Virtually all programming languages use whitespace as indentation to improve readability. In fact, if you look at some code, any code, you'll see that there's a strong correlation between your indentations and the code's branches and loops. Our indentation-based metric calculates the number of indentations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



```
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. 4.50: Explaining whitespace complexity.

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.

All right, 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 of the codebase's context to interpret them.

4.2.6 X-Ray

X-Ray gives you Deep Insights into your Code

Hotspots are code that we have to work with frequently. 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? CodeScene's X-Ray feature answers these questions.

X-Ray is a language-dependent analysis. The supported programming languages are listed in the /usage/language-support section.

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.

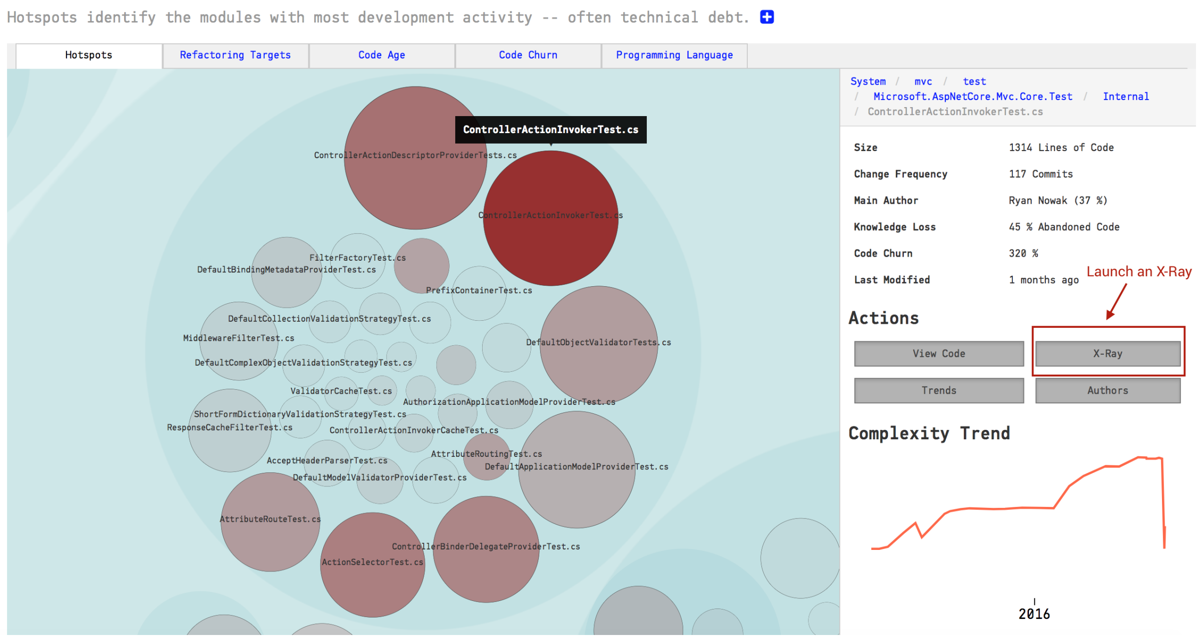
In the following guide we'll cover all of these cases. Let's start with how you can make sense of large files.

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. 4.51.


Hotspots identify the modules with most development activity -- often technical debt.

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

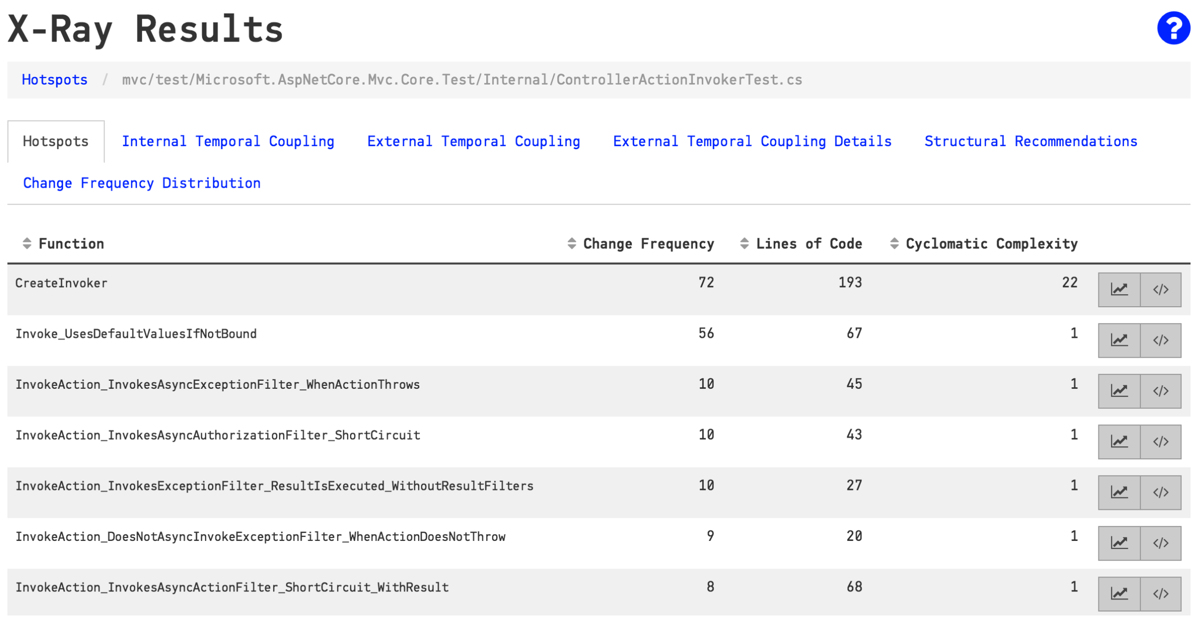
Size	1314 Lines of Code
Change Frequency	117 Commits
Main Author	Ryan Nowak (37 %)
Knowledge Loss	45 % Abandoned Code
Code Churn	320 %
Last Modified	1 months ago

Hotspots
Refactoring Targets
Code Age
Code Churn
Programming Language
ControllerActionInvokerTest.cs
ControllerActionDescriptorProviderTests.cs
ControllerActionInvokerTest.cs
FilterFactoryTest.cs
DefaultBindingMetadataProviderTest.cs
PrefixContainerTest.cs
DefaultCollectionValidationStrategyTest.cs
MiddlewareFilterTest.cs
DefaultObjectValidatorTests.cs
DefaultComplexObjectValidationStrategyTest.cs
ValidatorCacheTest.cs
AuthorizationApplicationModelProviderTest.cs
ShortFormDictionaryValidationStrategyTest.cs
ResponseCacheFilterTest.cs
ControllerActionInvokerCacheTest.cs
AttributeRoutingTest.cs
AcceptHeaderParserTest.cs
DefaultApplicationModelProviderTest.cs
DefaultModelValidatorProviderTest.cs
AttributeRouteTest.cs
ControllerBinderDelegateProviderTest.cs
ActionSelectorTest.cs
Launch an X-Ray
Actions
View Code
X-Ray
Trends
Authors
Complexity Trend
2016

Fig. 4.51: 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. 4.52 as a starting-point.



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	Lines of Code	Cyclomatic Complexity
CreateInvoker	72	193	22
Invoke_UsesDefaultValuesIfNotBound	56	67	1
InvokeAction_InvokesAsyncExceptionFilter_WhenActionThrows	10	45	1
InvokeAction_InvokesAsyncAuthorizationFilter_ShortCircuit	10	43	1
InvokeAction_InvokesExceptionFilter_ResultIsExecuted_WithoutResultFilters	10	27	1
InvokeAction_DoesNotAsyncInvokeExceptionFilter_WhenActionDoesNotThrow	9	20	1
InvokeAction_InvokesAsyncActionFilter_ShortCircuit_WithResult	8	68	1

Fig. 4.52: The starting point in an X-Ray analysis.

Fig. 4.52 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. You also see that *CreateInvoker* has a Cyclomatic Complexity of 22, which is a fairly high number. Thus, the method represents complicated code that you also have to work with often.

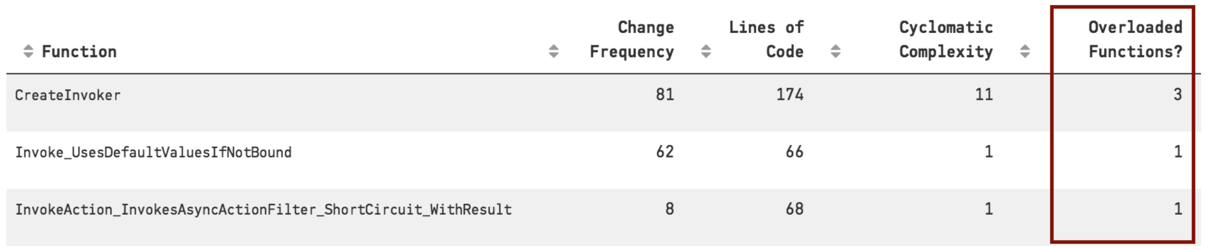
Methods like this are exactly where you'd like to focus your refactoring efforts; The high change frequency of the method indicates that improvements are likely to pay-off immediately. And the lines of code and complexity numbers gives you a sense of the effort you need to invest to make the necessary improvements.

But X-Ray gives you more information. As you see in the table above, CodeScene also 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 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.

A Note on Overloaded Methods

Some languages like C++, C#, and Java let you use the same function name for different implementations. In that case, X-Ray will combine all overloads with the same name into a single unit of measure. That is, if you have functions with the signature *f(int)* and *f(string)* they will be combined in the analysis. This approach typically gives you better results since the overloaded functions are part of the same logical unit of design and you want to analyze them as such.

CodeScene includes a count on the total number of methods to highlight such overloads, as shown in Fig. 4.53.



Function	Change Frequency	Lines of Code	Cyclomatic Complexity	Overloaded Functions?
CreateInvoker	81	174	11	3
Invoke_UsesDefaultValuesIfNotBound	62	66	1	1
InvokeAction_InvokesAsyncActionFilter_ShortCircuit_WithResult	8	68	1	1

Fig. 4.53: X-Ray highlights the total number of methods behind each overloaded hotspot.

Interpret Cyclomatic Complexity as part of the Evolutionary Metrics

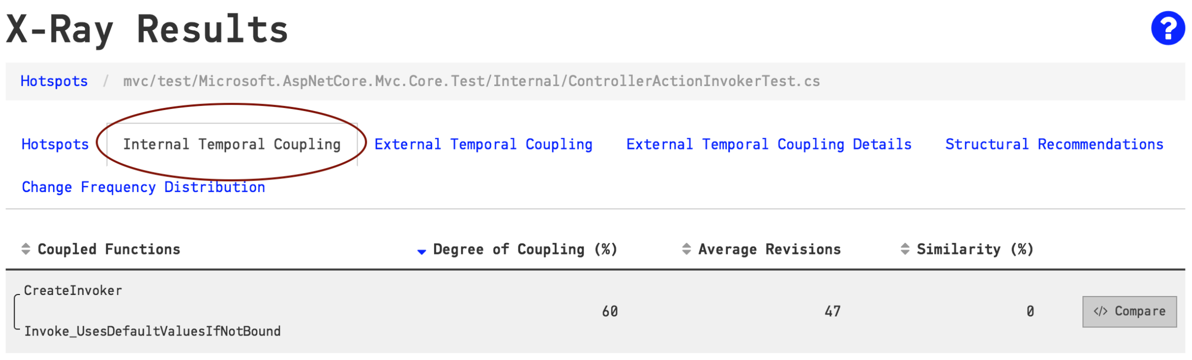
The cyclomatic complexity measure included in X-Ray doesn't stand on its own. Just because some code is complex doesn't mean it's a problem. However, when we combine a complexity measure with change frequencies – like X-Ray does – we get information we can act upon since the code complexity is put into context.

CodeScene includes its cyclomatic complexity metric as a supplement to the other information as a decent approximation of code quality. As a rule of thumb, any cyclomatic complexity value above 10 is likely to be problematic. A cyclomatic complexity beyond 25 is likely to hint at a true maintenance nightmare. But again, use the complexity value as a guide, not as an absolute truth.

Cyclomatic complexity also helps you make refactoring decisions in the sense that you get a rough idea on how hard the code will be to test. Each branch in your functions add to their complexity value and, as a direct consequence, to the testing efforts.

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. 4.54.


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
Compare

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

Fig. 4.54: X-Ray calculates temporal coupling between the methods in your Hotspot.

Fig. 4.54 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. 4.55.

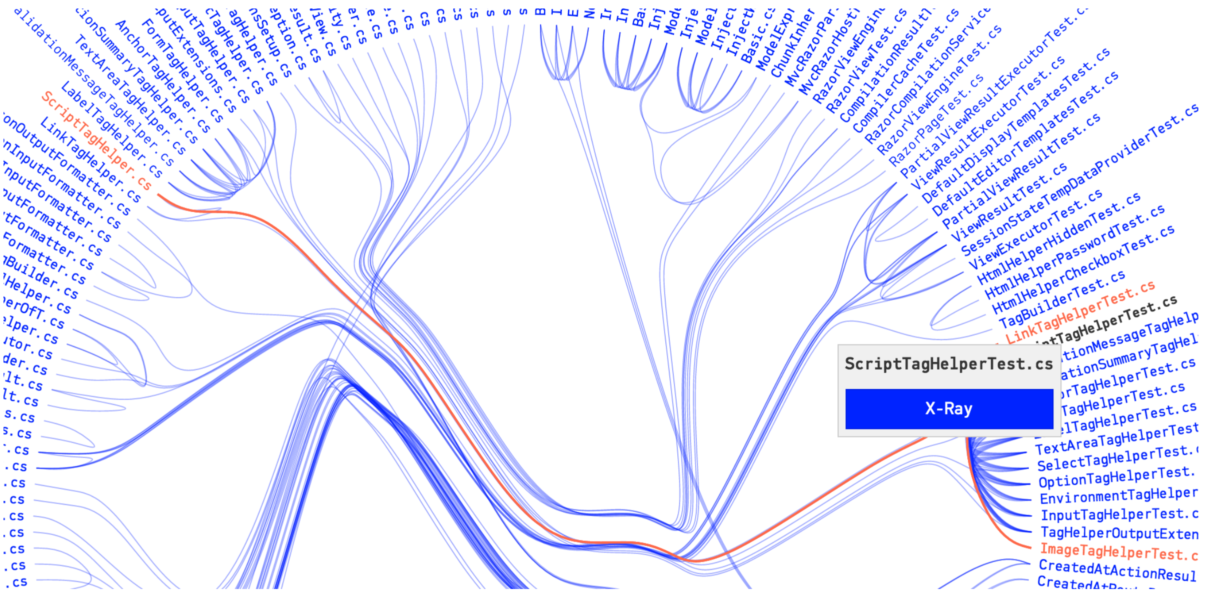


Fig. 4.55: X-Ray lets you investigate temporal coupling clusters in detail.

Once X-Ray is done, you're presented with a dependency wheel on method level. Have a look the dependency wheel in Fig. 4.56 and I'll walk you though the details.

The dependency wheel in Fig. 4.56 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.

Find change patterns across repository boundaries

Since CodeScene's analyses are language neutral it can identify implicit/hidden change patterns between code implemented in different languages. But CodeScene can go an extra mile: it can even uncover such change patterns when the different files are located in separate Git repositories! Take a look at the X-Ray results in Fig. 4.57.

As you see in the preceding figure, X-Ray works across Git repository boundaries to identify the functions responsible for the temporal coupling. This is a powerful analysis that is particularly useful to:

- *Microservices*: Implicit dependencies across service boundaries is problematic since it couples the life cycle of different services to each other. Use CodeScene to detect and X-Ray such dependencies.
- *Producer/Consumer*: The preceding example is a modern variation of the client-server pattern. Use X-Ray to learn about the change pattern in a complex, multi-repository project.
- *Inter-Team Coordination*: In large organizations different teams tend to be responsible for the code in different repositories. Using X-Ray's inter-repo analysis lets you uncover expensive change patterns that impact other teams.

Unfortunately X-Ray across repository boundaries doesn't work by magic; There has to be some mechanism to relate different commits to the same logical change set. CodeScene use Ticket IDs for that purpose, so all you need to do is to configure your Ticket ID patterns and this X-Ray feature will become enabled.

As a bonus, this feature also works well in the case of differing commit styles; Some organizations prefer to build their features by many small and incomplete commits. As a consequence, a single commit

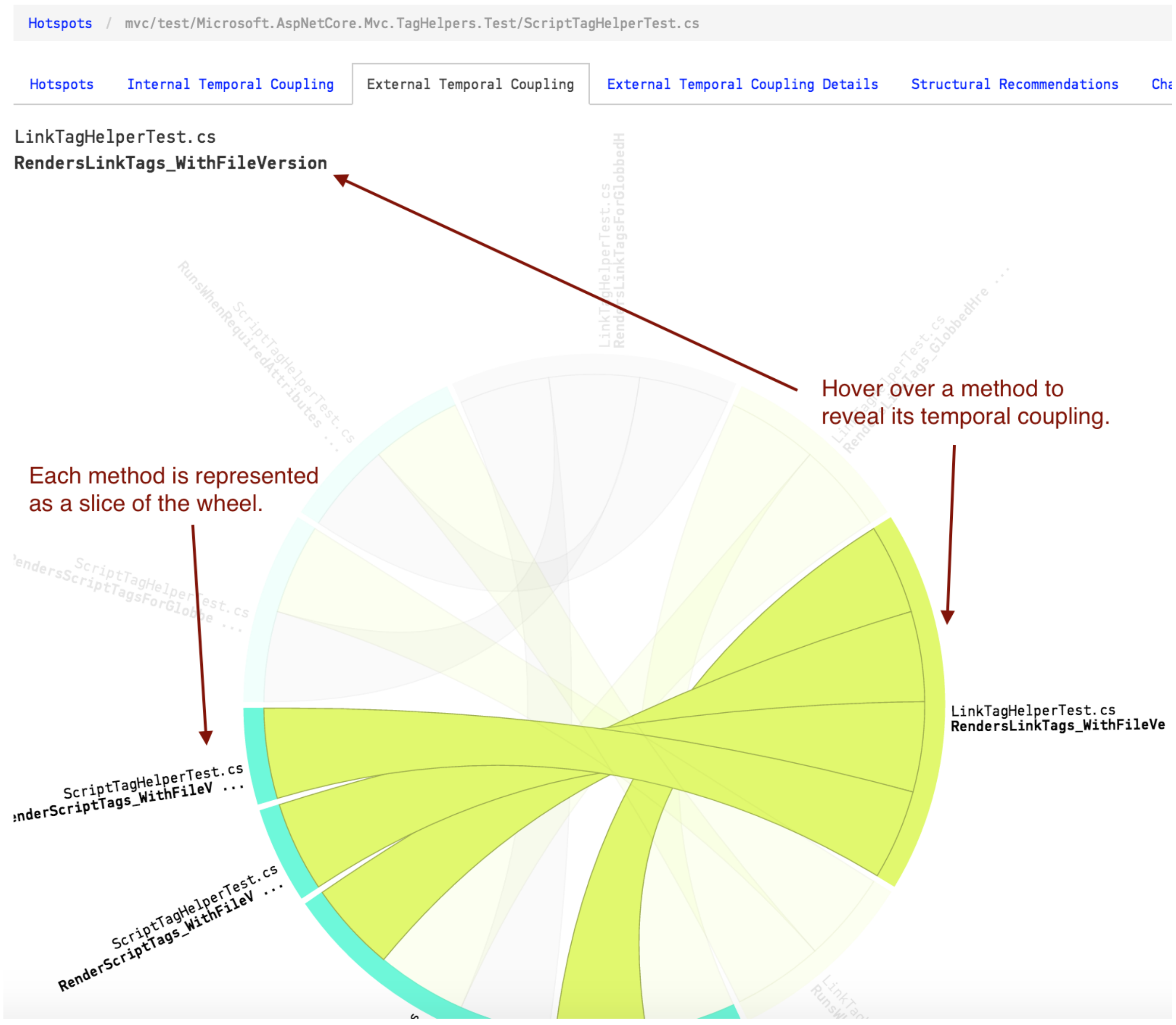

Hotspots / mvc/test/Microsoft.AspNetCore.Mvc.TagHelpers.Test/ScriptTagHelperTest.cs
Hotspots
Internal Temporal Coupling
External Temporal Coupling
External Temporal Coupling Details
Structural Recommendations
Cha
LinkTagHelperTest.cs
RendersLinkTags_WithFileVersion
Hover over a method to reveal its temporal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

Fig. 4.56: The dependency wheel shows the temporal coupling between methods.

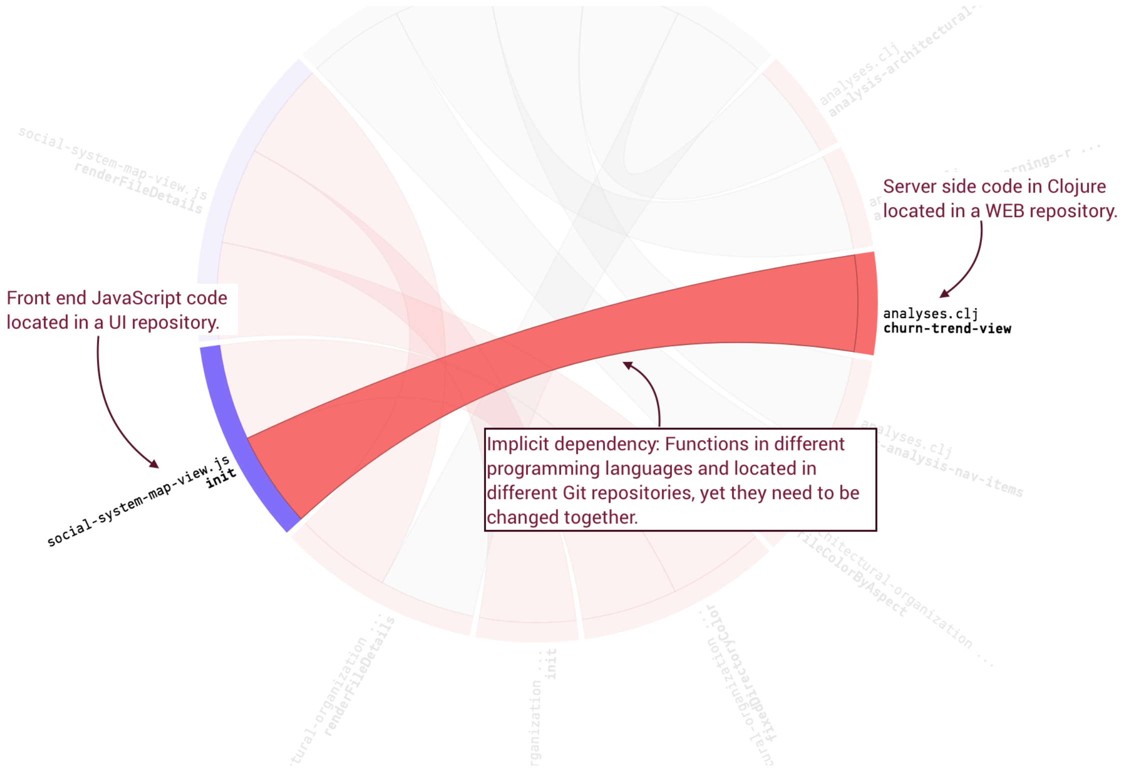

Server side code in Clojure located in a WEB repository.
analyses.clj
churn-trend-view
Front end JavaScript code located in a UI repository.
social-system-map-view.js
init
Implicit dependency: Functions in different programming languages and located in different Git repositories, yet they need to be changed together.

Fig. 4.57: X-Ray works across multiple repositories.

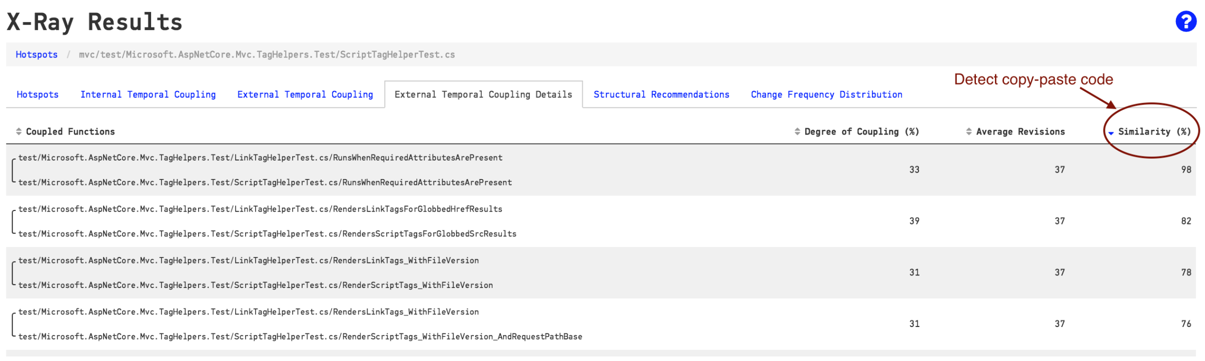
contains very little information and there's usually no temporal coupling between commits. Temporal coupling by Ticket ID provides a viable alternative here.

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. 4.58.


X-Ray Results
Hotspots / mvc/test/Microsoft.AspNetCore.Mvc.TagHelpers.Test/ScriptTagHelperTest.cs
Hotspots
Internal Temporal Coupling
External Temporal Coupling
External Temporal Coupling Details
Structural Recommendations
Change Frequency Distribution
Detect copy-paste code

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

In Fig. 4.58 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.

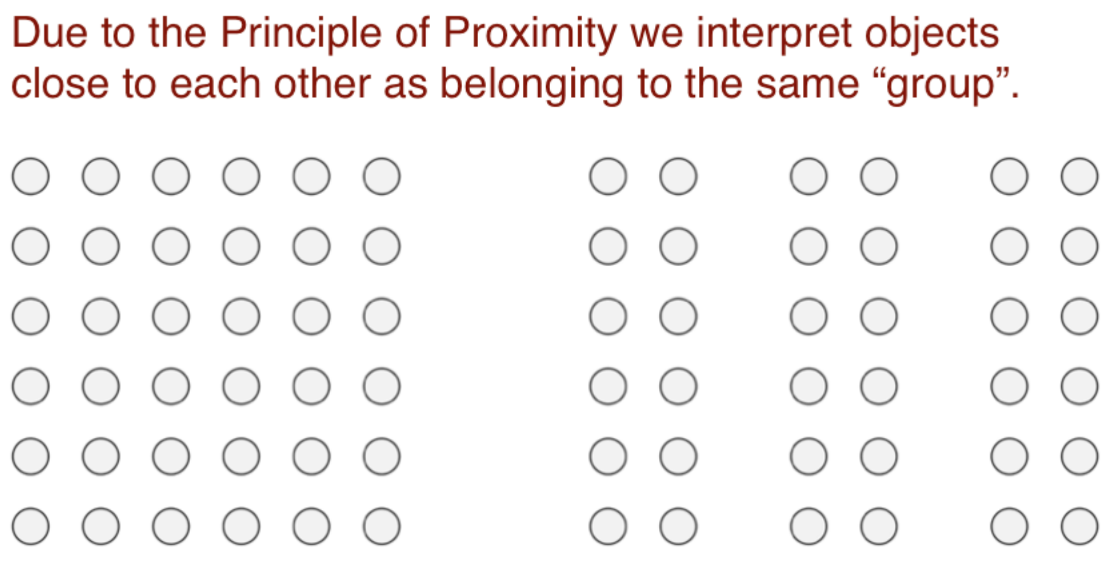
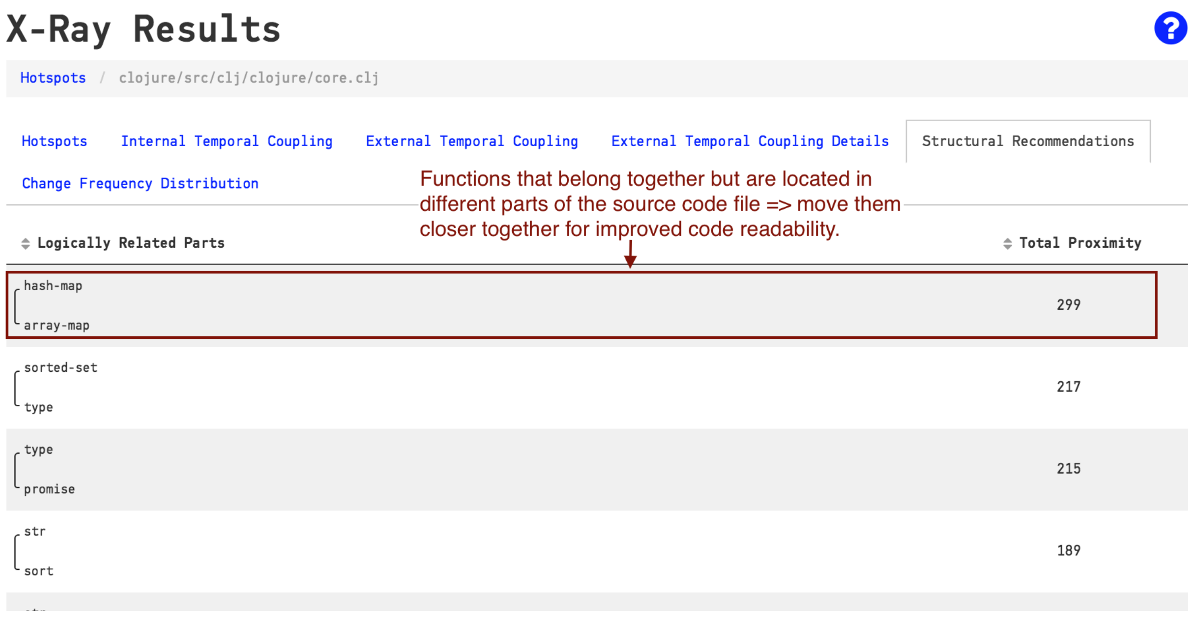

Due to the Principle of Proximity we interpret objects close to each other as belonging to the same "group".

Fig. 4.59: An illustration of the Principle of Proximity where our brain forms groups of related objects.

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. 4.59. 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. 4.60 from the implementation of the Clojure programming language.


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

The highlighted recommendation in Fig. 4.60 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. 4.60 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. 4.60 shows that there's a total proximity of *299*. That means that 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).

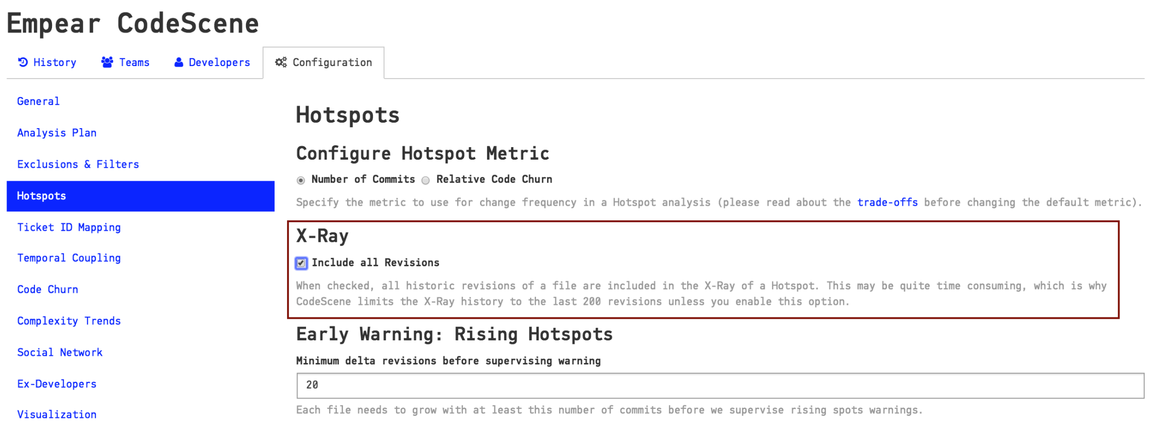
So if you want to ensure that your renamed methods are being tracked past the rename, please make sure that you do the renaming in one commit and possible method body modifications in another commit. It's usually a good refactoring practice anyway.

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. 4.61.



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

4.2.7 Development Output and Code Churn

CodeScene's development output metrics indicates the rate at which your code evolves. Some common use cases include:

- *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.

- *Track trends by task:* CodeScene lets you inspect the size and impact of your tasks. Use the information to see if your project management tasks are on an appropriate level or if each one of them implies a mini big bang in terms of code changes.

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. 4.62.

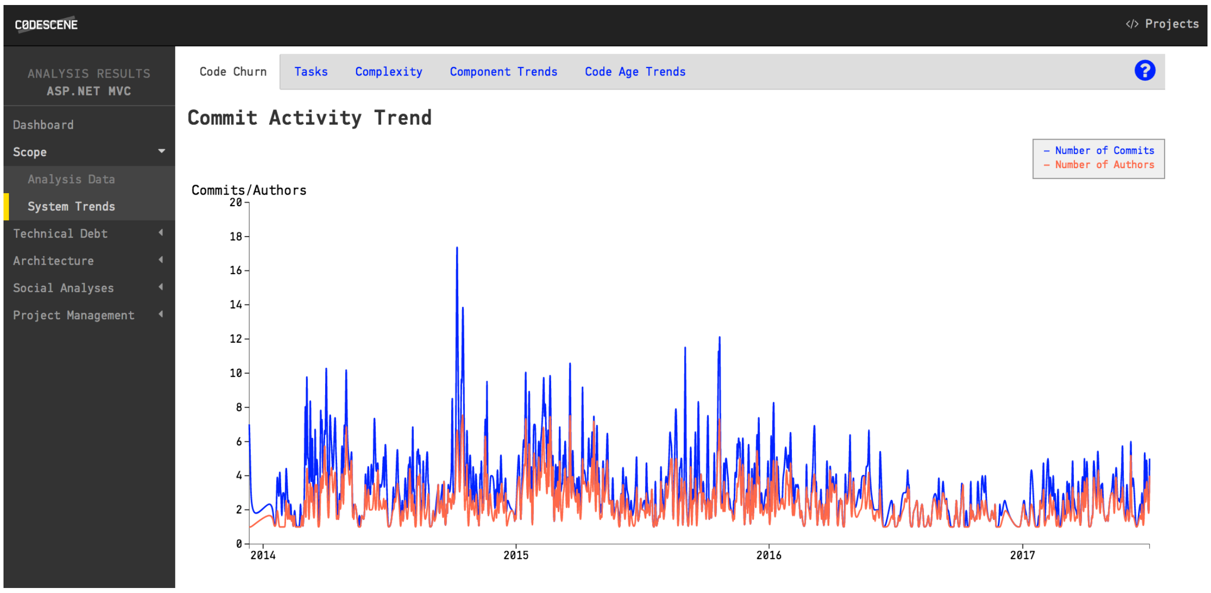


Fig. 4.62: The commit activity chart.

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.

Correlate Trends in the Number of Contributors with the Churn Metrics

The Active Contributors trend shows the number of authors in the codebase over time. CodeScene calculates this information by looking at the first and last recorded contribution times for each author. This lets you view a trend as illustrated in Fig. 4.63.

The contribution trend is particularly interesting when correlated with the other churn trends. You may also want to compare the contribution trend with the system complexity trend (see *Architectural Analyses* (page 88)). Correlating the number of active contributors to these churn metrics lets you evaluate the effect (or lack thereof) when a project is scaled up or down.

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.

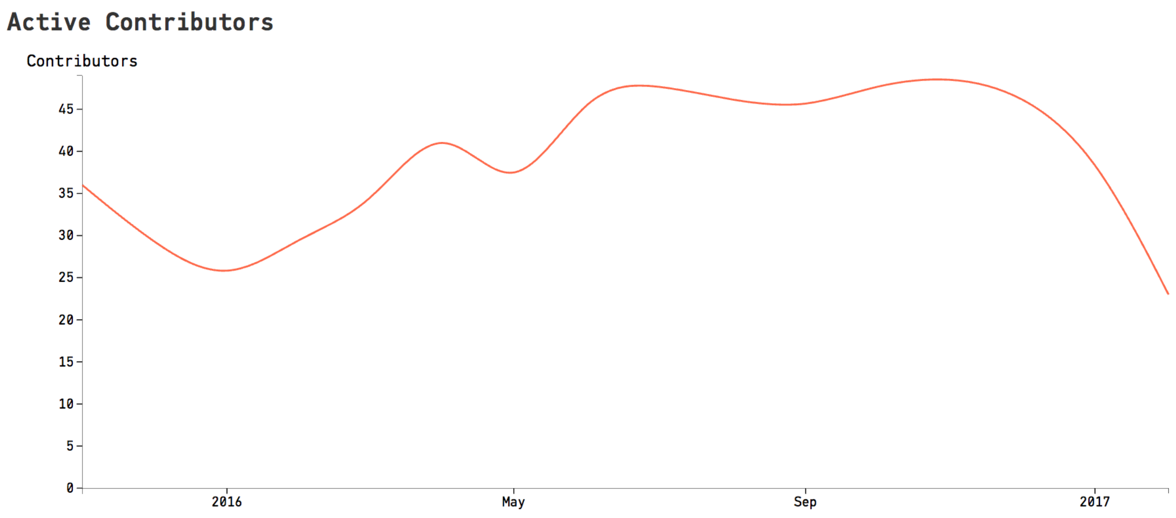


Fig. 4.63: The number of active contributors over time.

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. 4.64.

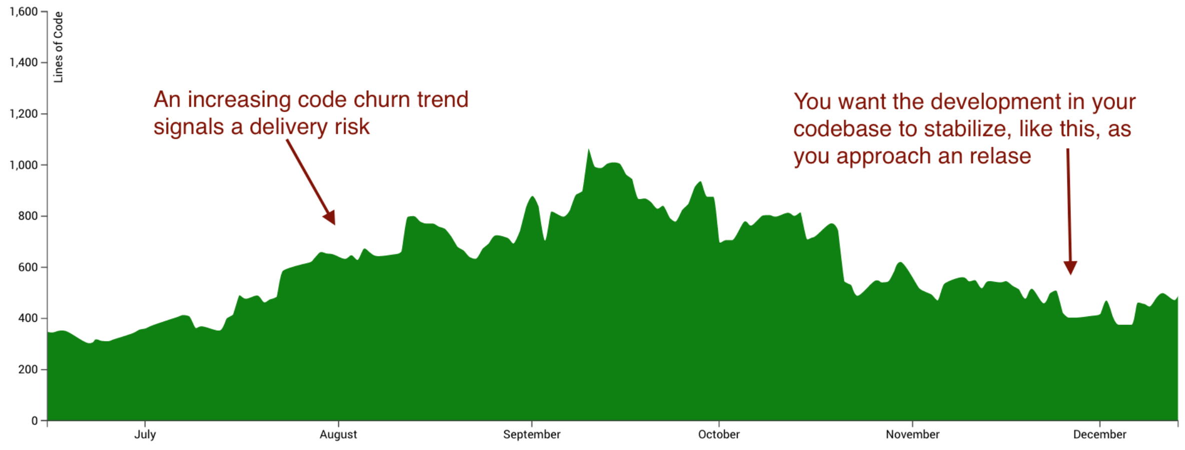


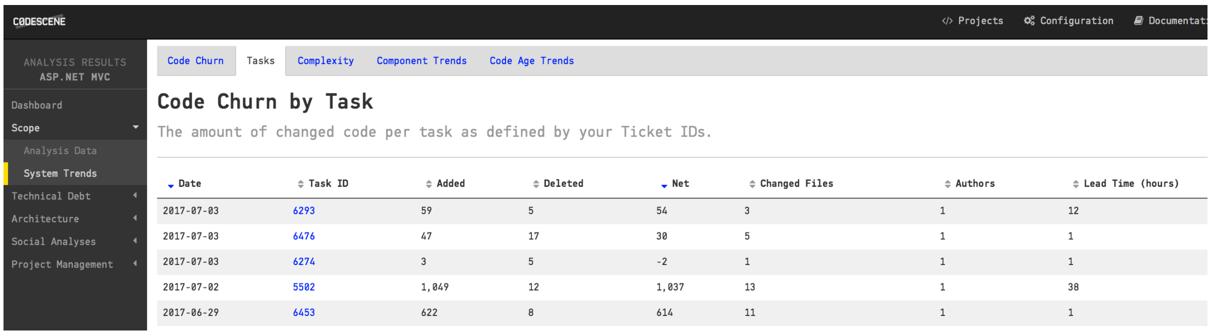
Fig. 4.64: Use code churn to reason about delivery risk.

Inspect The Level of your Tasks

It's a challenge to strike the right level when you partition your work into individual tasks. Large tasks are hard to reason about and also less predictable to plan. That's why we generally prefer small and well-focused tasks.

CodeScene lets you inspect the impact of your project management tasks on the codebase in terms of both code churn and collaboration (that is, how many authors worked together to solve the task?) as illustrated in Fig. 4.65.

CodeScene uses your Ticket ID to group individual commits into tasks. As you see in Fig. 4.65, CodeScene also calculates a *Lead Time* for each task. The lead time is the time that passed between the first and the last commit referencing a specific task. Note that many tasks tend to be solved in a single commit. In that case CodeScene doesn't have the data to calculate a proper lead time. So CodeScene defaults to one hour in case of a task that's resolved in a single commit.



Code Churn by Task

The amount of changed code per task as defined by your Ticket IDs.

Date	Task ID	Added	Deleted	Net	Changed Files	Authors	Lead Time (hours)
2017-07-03	6293	59	5	54	3	1	12
2017-07-03	6476	47	17	30	5	1	1
2017-07-03	6274	3	5	-2	1	1	1
2017-07-02	5502	1,049	12	1,037	13	1	38
2017-06-29	6453	622	8	614	11	1	1

Fig. 4.65: Inspect the code churn on a task level.

We recommend that you use the lead time data to track tasks that drift in time. Often, those tasks suggests requirements that aren't as well-defined as they could be. You can also use the lead time analysis to track the effects of process changes in your organization.

4.2.8 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 37)), 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. 4.66 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.

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

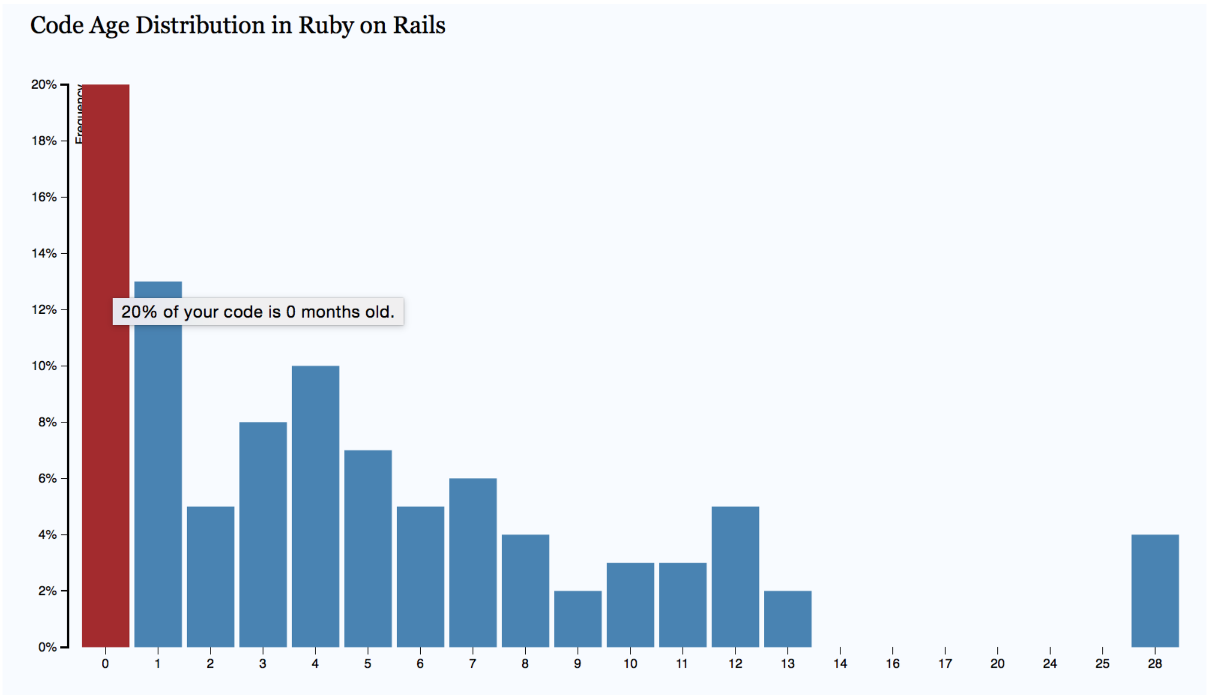


Fig. 4.66: An example of code age distribution.

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. 4.67.

This is a good starting point; If you have a lot of code, as in the distribution in Fig. 4.67, 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.

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.

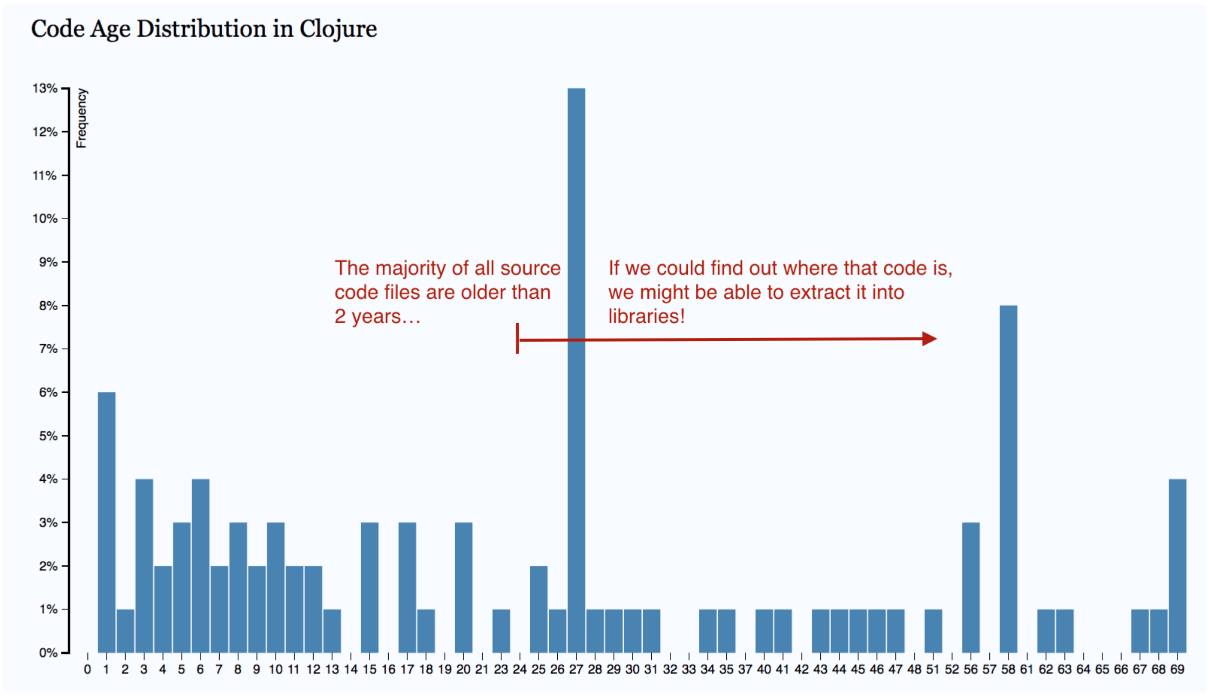


Fig. 4.67: Code age distribution in Clojure.

Fig. 4.68: Annual rings of a tree.

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. 4.69 hows how we find that code.

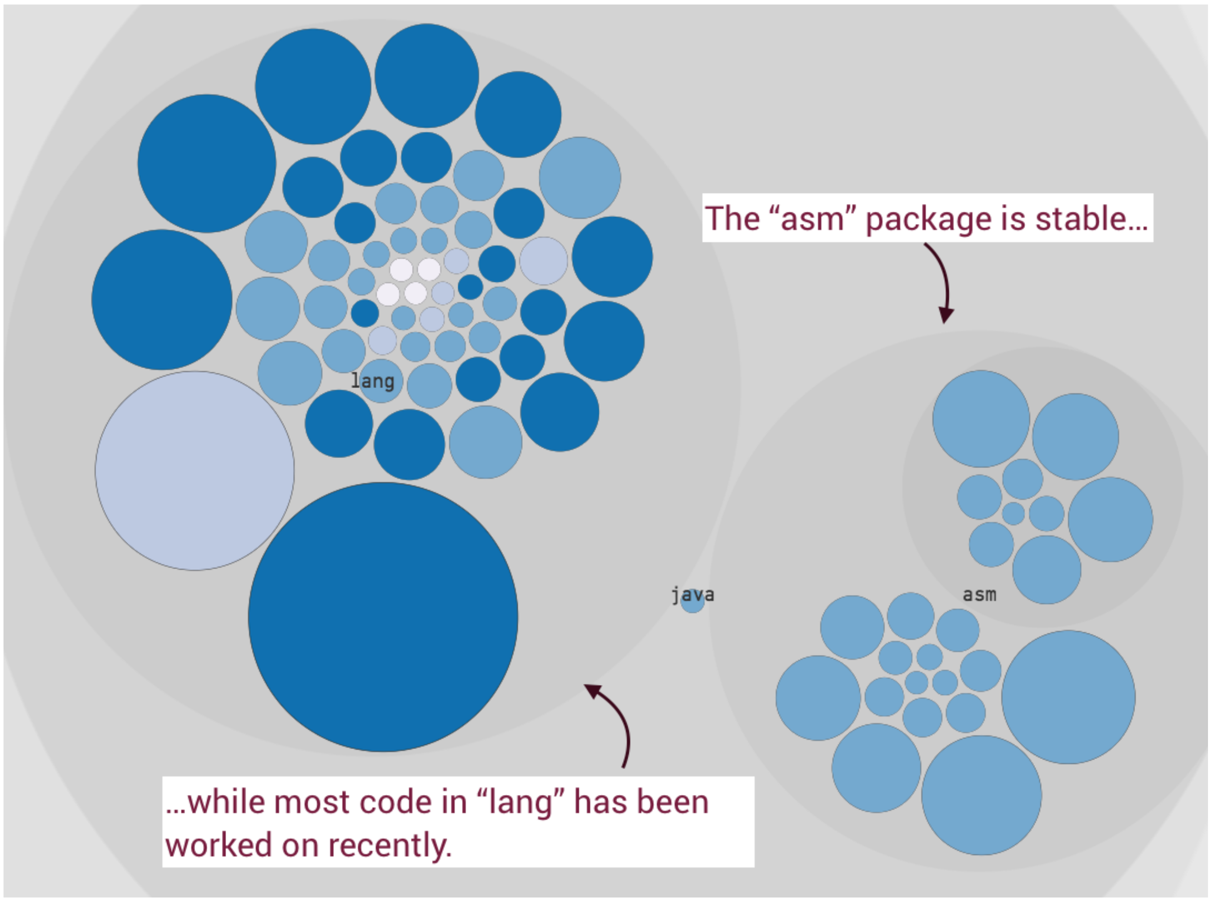

The "asm" package is stable...
lang
java
asm
...while most code in "lang" has been worked on recently.

Fig. 4.69: Stable code in Clojure.

Extract Stable Packages as Libraries

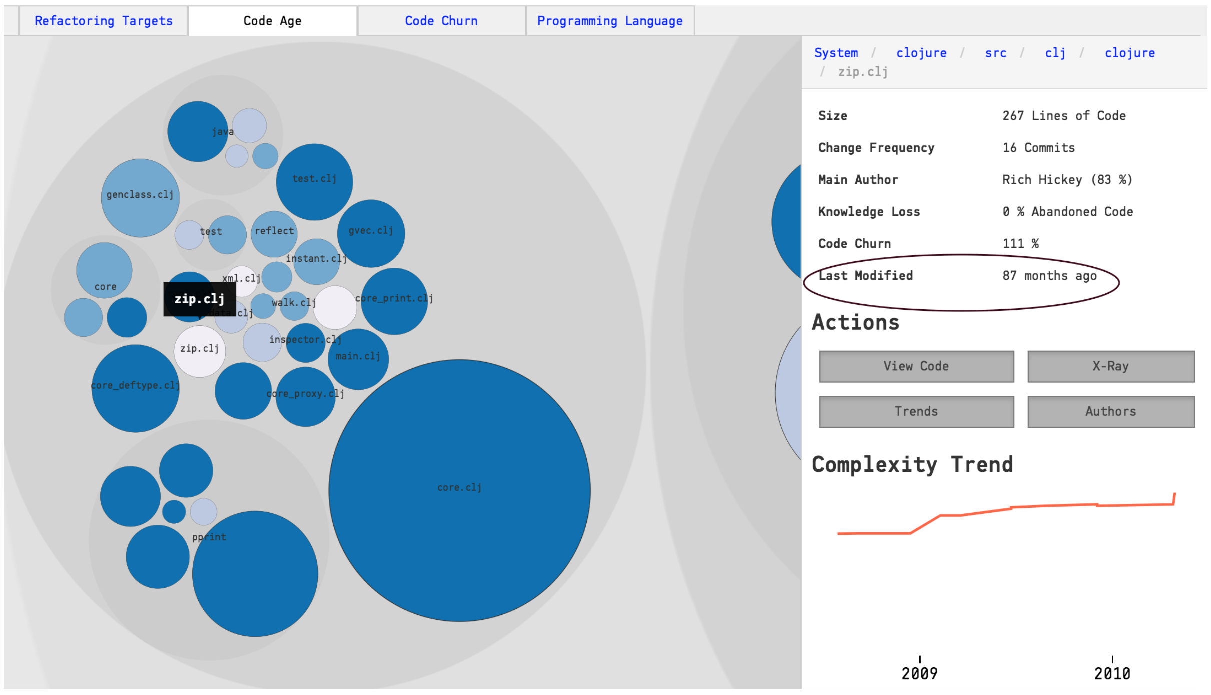
Once you've found stable packages you may want to consider to extract them as packages. If we transform stable packages into libraries we get a set of advantages:

- *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. 4.70



Size	267 Lines of Code
Change Frequency	16 Commits
Main Author	Rich Hickey (83 %)
Knowledge Loss	0 % Abandoned Code
Code Churn	111 %
Last Modified	87 months ago

Fig. 4.70: 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.

4.3 Architectural

4.3.1 Architectural Analyses

CodeScene's architectural analyses lets you run Hotspots, Temporal Coupling, Code Health, and more at 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.

With CodeScene, you get the same information on an architectural level as the file level analyses, as illustrated in Fig. 4.71. Note that this is information that isn't available in your code.

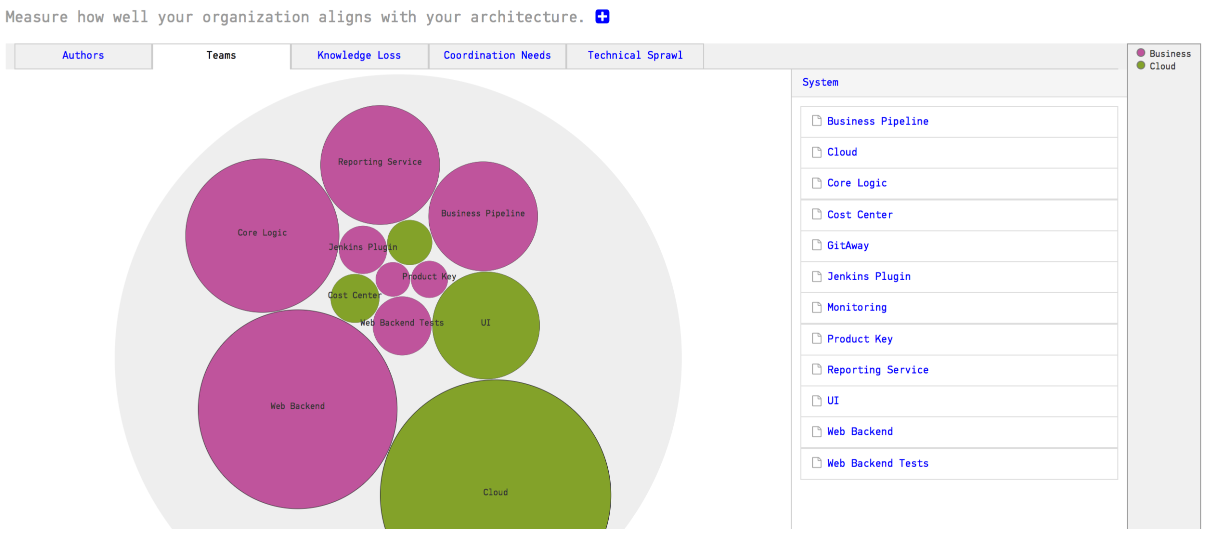
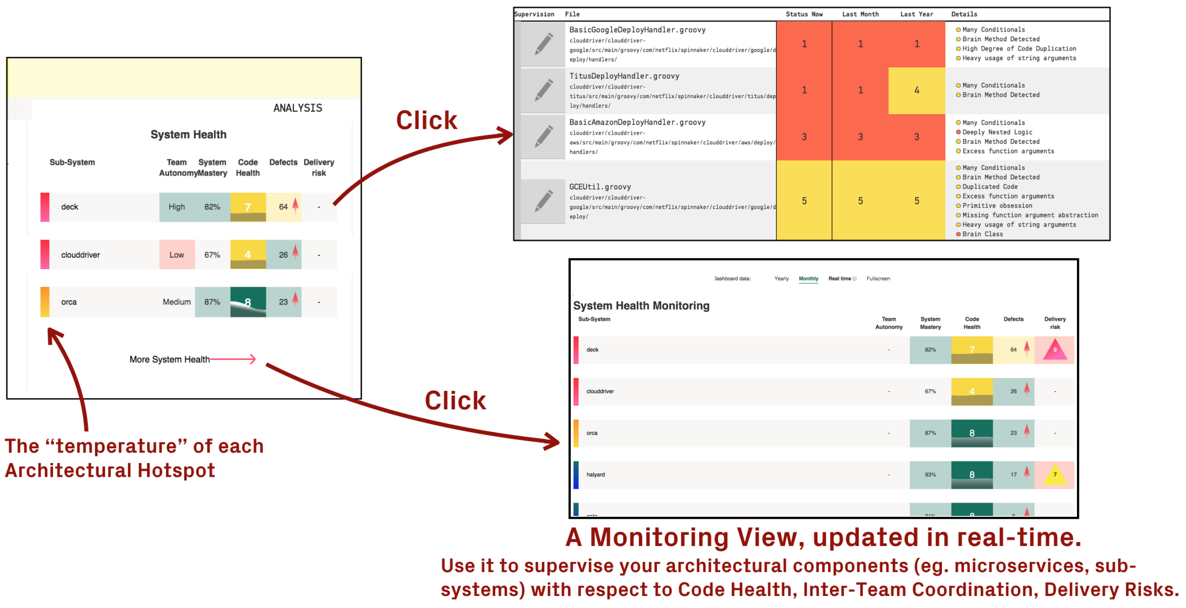


Fig. 4.71: High level architectural analyses on the technical and social aspects of code.

By enabling the architectural analyses, you also get a System Hotspots Health dashboad that shows you the high-level metrics for each one of your architectural components or microservices.


Click
Click
The "temperature" of each Architectural Hotspot
A Monitoring View, updated in real-time.
Use it to supervise your architectural components (eg. microservices, sub-systems) with respect to Code Health, Inter-Team Coordination, Delivery Risks.
ANALYSIS
More System Health
Dashboard data: Yearly Monthly Real time Fullscreen

System Health

Sub-System	Team Autonomy	System Mastery	Code Health	Defects	Delivery risk
deck	High	82%	7	64	-
clouddriver	Low	67%	4	26	-
orca	Medium	87%	8	23	-

Supervision	File	Status Now	Last Month	Last Year	Details
	BasicGoogleDeployHandler.groovy clouddriver/clouddriver-google/src/main/groovy/com/netflix/spinnaker/clouddriver/google/deploy/handlers/	1	1	1	Many Conditionals; Brain Method Detected; High Degree of Code Duplication; Heavy usage of string arguments
	TitusDeployHandler.groovy clouddriver/clouddriver-titus/src/main/groovy/com/netflix/spinnaker/clouddriver/titus/deploy/handlers/	1	1	4	Many Conditionals; Brain Method Detected
	BasicAmazonDeployHandler.groovy clouddriver/clouddriver-aws/src/main/groovy/com/netflix/spinnaker/clouddriver/aws/deploy/handlers/	3	3	3	Many Conditionals; Deeply Nested Logic; Brain Method Detected; Excess function arguments
	GCEUtil.groovy clouddriver/clouddriver-google/src/main/groovy/com/netflix/spinnaker/clouddriver/google/deploy/	5	5	5	Many Conditionals; Brain Method Detected; Duplicated Code; Excess function arguments; Primitive obsession; Missing function argument abstraction; Heavy usage of string arguments; Brain Class

System Health Monitoring

Sub-System	Team Autonomy	System Mastery	Code Health	Defects	Delivery risk
deck	-	82%	7	64	9
clouddriver	-	67%	4	26	-
orca	-	87%	8	23	-
halyard	-	93%	8	17	7

Fig. 4.72: The System Health dashboard lets you monitor the evolution of your sub-systems, services, or microservices.

This section of the guide walks you through the necessary configuration and gives you some ideas on how to interpret and act upon the architectural analysis results.

What is 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.

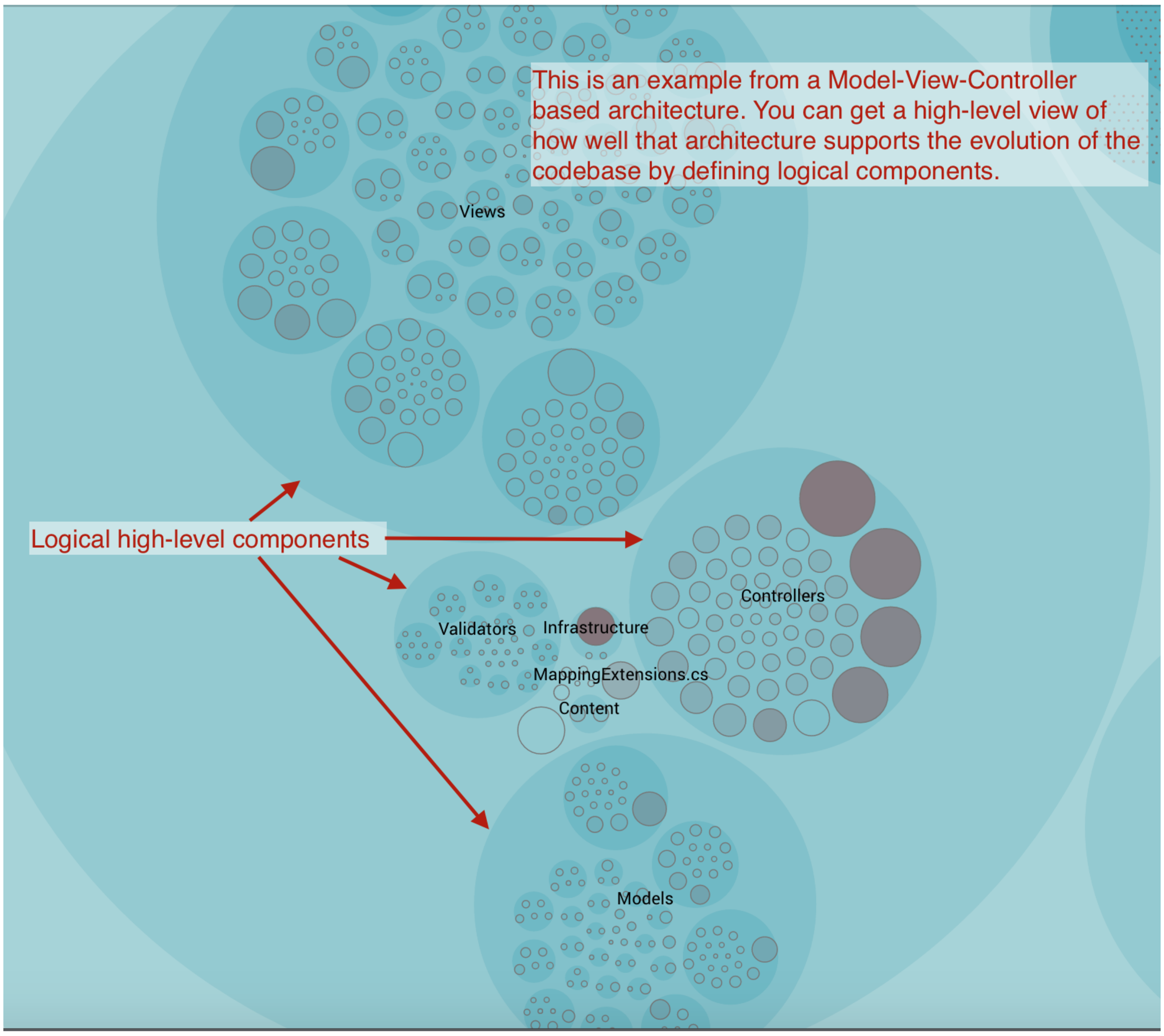
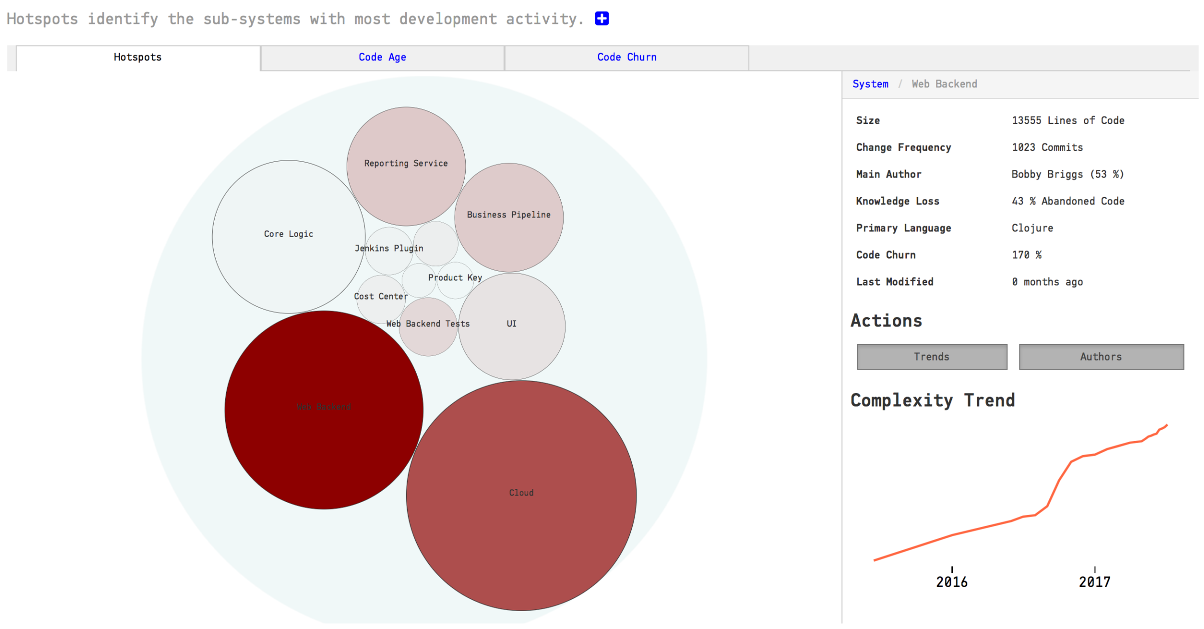

This is an example from a Model-View-Controller based architecture. You can get a high-level view of how well that architecture supports the evolution of the codebase by defining logical components.
Views
Logical high-level components
Controllers
Validators
Infrastructure
MappingExtensions.cs
Content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

Fig. 4.73: 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*.

You'll learn to define your components in the next section. Before we go there, let's have a look at the end result.

As you see in Fig. 4.74, CodeScene presents a hotspot analysis on architectural level. This gives you a high-level view of how your development activity is focused. You also see that you get the social knowledge metrics on an architectural level too. We'll discuss that in more detail later in this guide to learn how we use them to analyse complex architectures like Microservices.


Hotspots identify the sub-systems with most development activity.

System / Web Backend

Size	13555 Lines of Code
Change Frequency	1023 Commits
Main Author	Bobby Briggs (53 %)
Knowledge Loss	43 % Abandoned Code
Primary Language	Clojure
Code Churn	170 %
Last Modified	0 months ago

Actions

Trends Authors

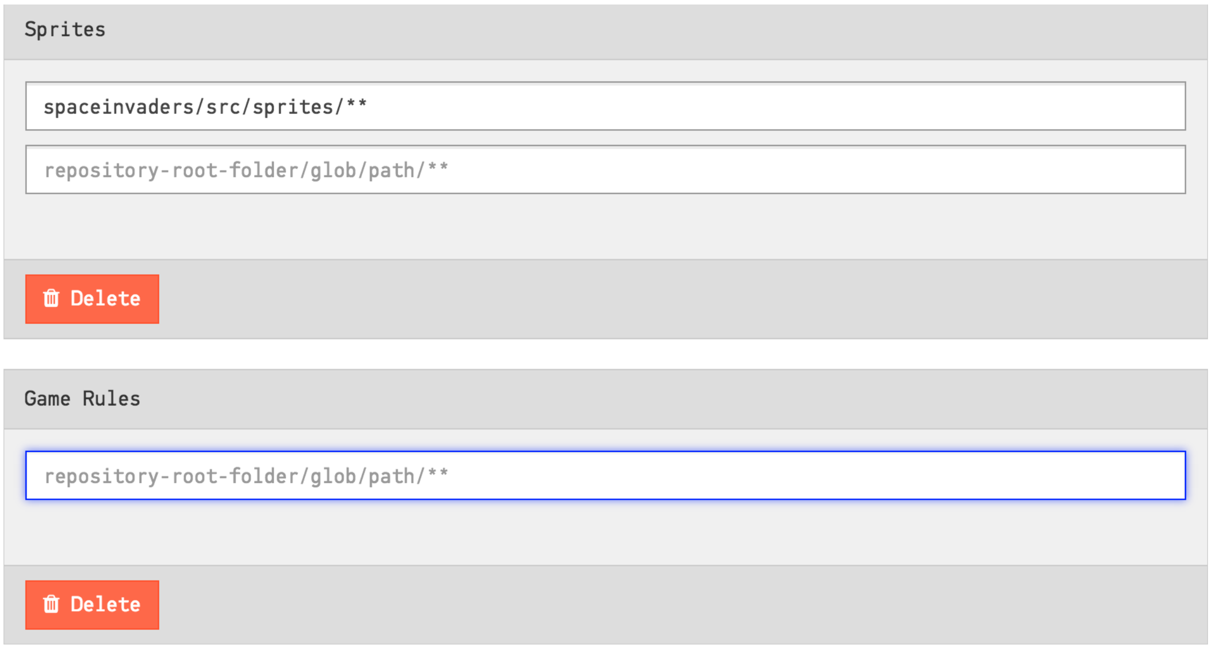
Complexity Trend

Fig. 4.74: High level architectural hotspots analysis.

Define your Architectural Components

You need to configure your Architectural Components in order to enable these analyses, and the Architectural Components you specify depend upon your architectural style. You may also 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.

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. 4.75.


Sprites

spaceinvaders/src/sprites/**

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

Delete

Game Rules

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

Delete

Fig. 4.75: 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.

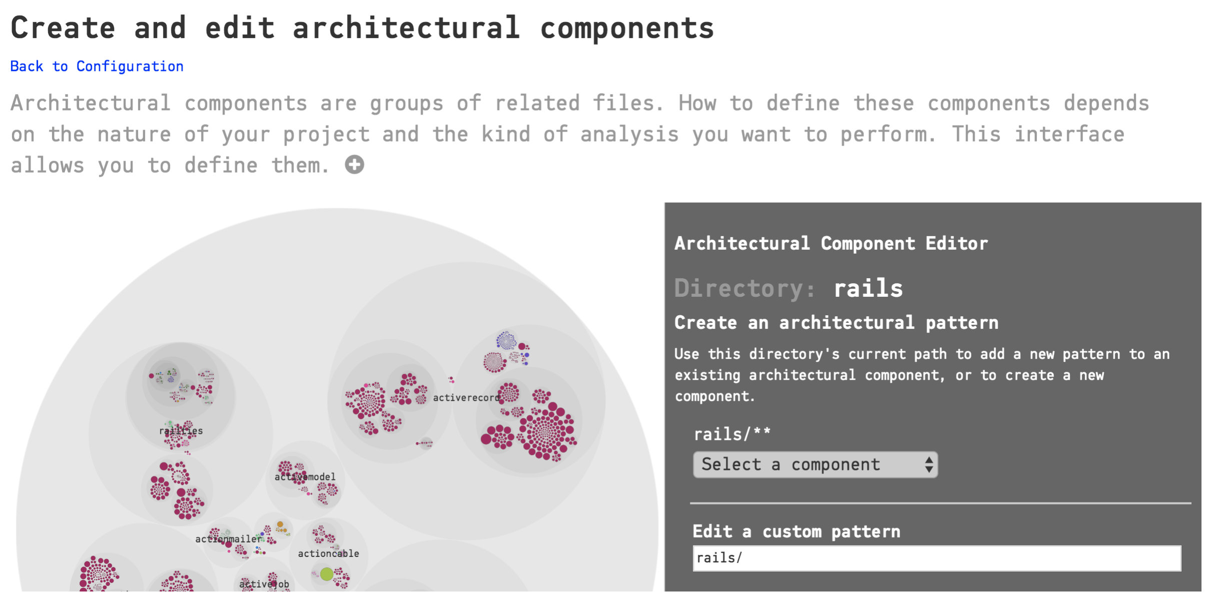
Let's consider the example above to learn more about the format. The configuration in Fig. 4.75 specifies the pattern *spaceinvaders/source/sprites/***. That means that all content under the folder *spaceinvaders/source/sprites* will be considered as the architectural/logical component *Sprites*.

In general, you want to match architectural components on the level of the different sub-folders of your codebase. But you can of course provide much more granular filters and, with the power of glob patterns, match all files sharing a common extension, or even individual files.

You can also map multiple folders to the same architectural component. A common example on this is when you want to consider the application code and its associated unit tests as one logical unit. In this case you'd add a second pattern to the *Sprites* component in Fig. 4.75: *spaceinvaders/test/sprites/***.

Use the Architectural Component Editor

The most common way of defining your architectural components is to use the Architectural Component Editor. In the "Architecture" tab of your project's configuration pages, a large button leads to the Editor.



Create and edit architectural components

Back to Configuration

Architectural components are groups of related files. How to define these components depends on the nature of your project and the kind of analysis you want to perform. This interface allows you to define them.

Architectural Component Editor

Directory: **rails**

Create an architectural pattern

Use this directory's current path to add a new pattern to an existing architectural component, or to create a new component.

rails/**

Select a component

Edit a custom pattern

rails/

Fig. 4.76: CodeScene's Architectural Component Editor provides a visual interface to your project's files.

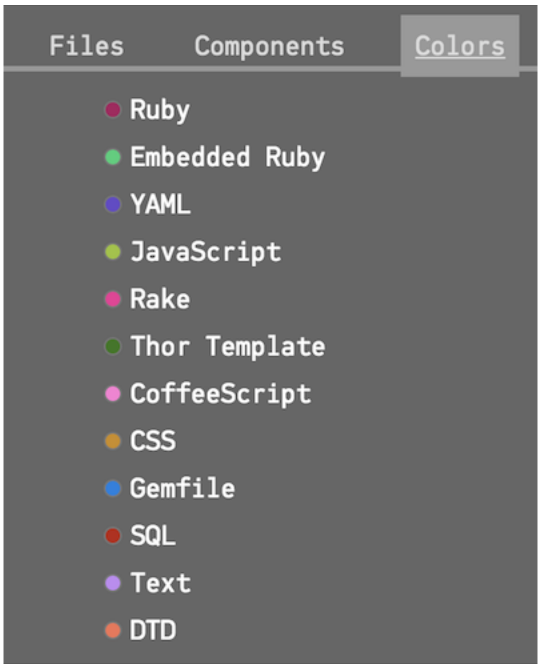
The Editor provides a visual interface to the files in your project. For this reason, it can only be used after you run an initial analysis. Once CodeScene is aware of the files in your project, it will provide you with the same circular visualization used for Hotspots and other analyses. You can zoom in and out to choose the parts of your project that you want to include in a Component. The colors of the circles indicate the type of files. A legend is available in one of the tabs of the sidebar:

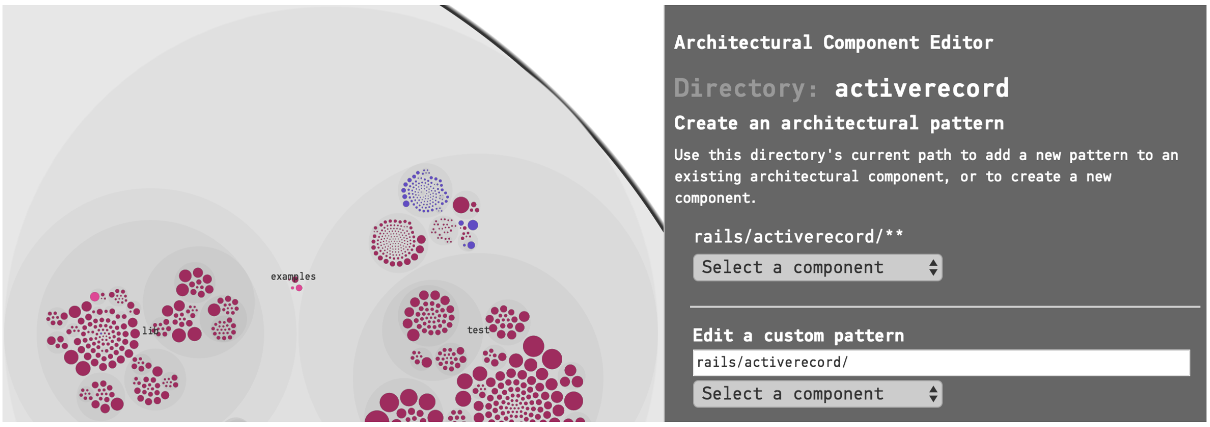
When you have located a directory or a file that you wish to include in a Component, you have two choices at the top of the sidebar on the right:

The most common action here is to click on "Select a component" under the first pattern, which, in the example above is *rails/activerecord/***. This pattern will match all the files and subdirectories in the *activerecord* directory. You can either add the pattern to an existing Component, or create a new Component based on your selection.

The other choice is to write a custom pattern. In this example, if we were only interested in the *.yml* files in the *activerecord* directory, we could create a pattern like this:

This way, while using the visual interface, you still have the full power of glob patterns. Note that patterns are validated and must begin with the project root of the corresponding Git repository (*rails/* in this example). or with ***. The interface will prevent you from entering invalid patterns.





Architectural Component Editor

Directory: activerecord

Create an architectural pattern

Use this directory's current path to add a new pattern to an existing architectural component, or to create a new component.

rails/activerecord/**

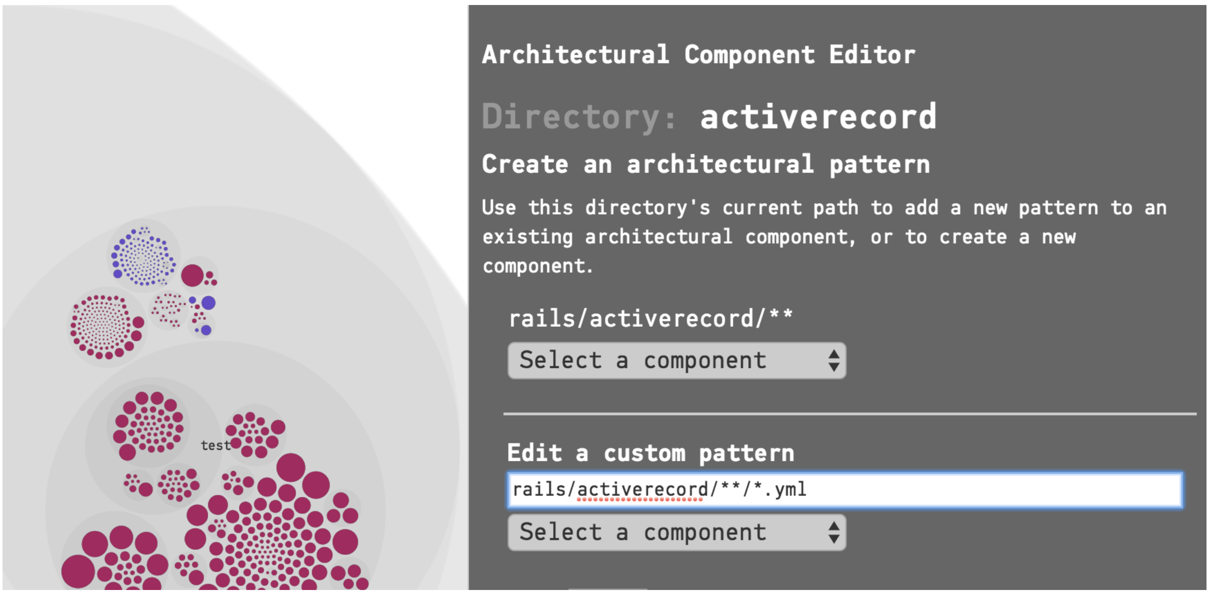
Select a component

Edit a custom pattern

rails/activerecord/

Select a component

Fig. 4.77: Choose either the pattern for the current directory, or write your own pattern.



Architectural Component Editor

Directory: activerecord

Create an architectural pattern

Use this directory's current path to add a new pattern to an existing architectural component, or to create a new component.

rails/activerecord/**

Select a component

Edit a custom pattern

rails/activerecord/**/*.yml

Select a component

Fig. 4.78: A custom pattern that selects all the .yml files inside a directory.

If you make a mistake, you can remove the pattern from the Component:

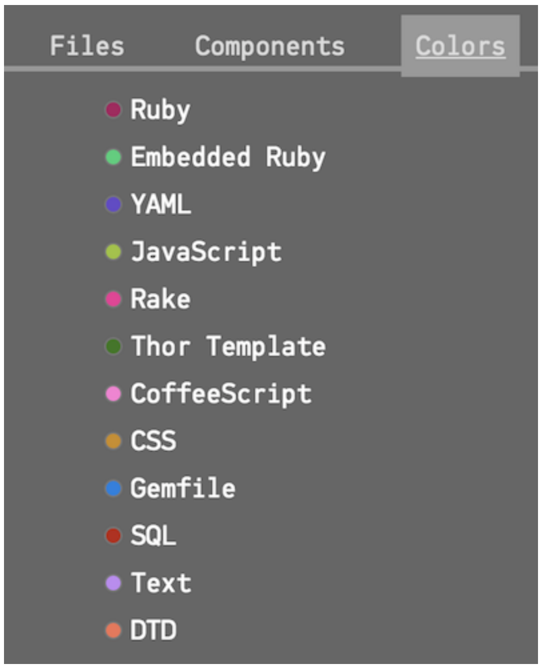
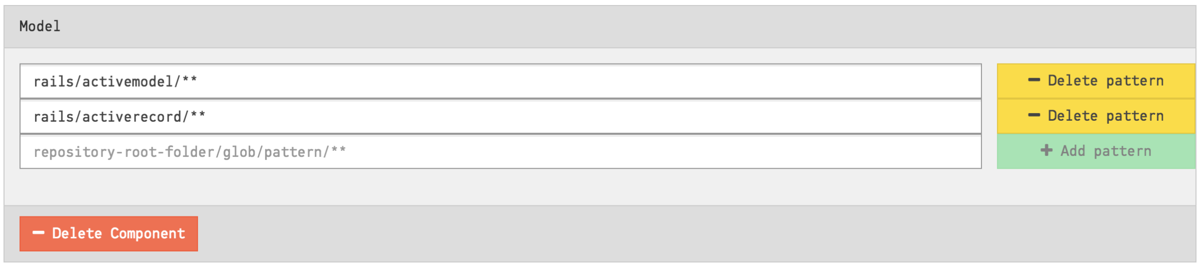


Fig. 4.79: Remove a pattern

The Architectural Component Editor also comprises a form-based view which you will find by scrolling further down the page.


Model

rails/activemodel/**

rails/activerecord/**

repository-root-folder/glob/pattern/**

Fig. 4.80: The Editor also has a form-based view

You can make changes here just like in the visual interface, adding, editing or deleting components and patterns. Note that when a component contains zero patterns, it is deleted.

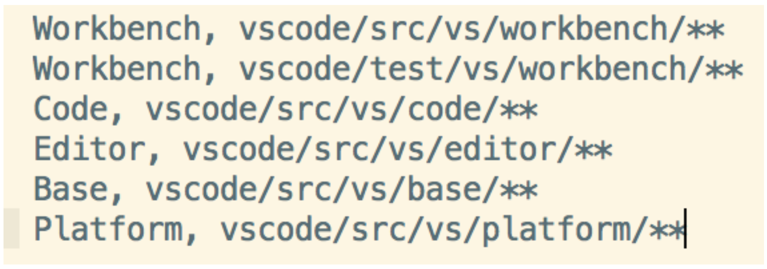
Changes are only stored when you click on the “Submit” button at the bottom of the page. Your new Architectural Components will be used the next time an analysis is run on your project.

Import or Export Architectural Component Definitions

Instead of specifying the patterns manually in the section above, you can import a CSV file with the definitions. This is a simpler option in a large system where you can script the generation of the CSV to import:

- Your CSV file must *not* include a header row.
- The CSV file shall contain two columns: 1. the Component Name and 2. its Glob Pattern.
- The fields in your CSV are separated by commas.

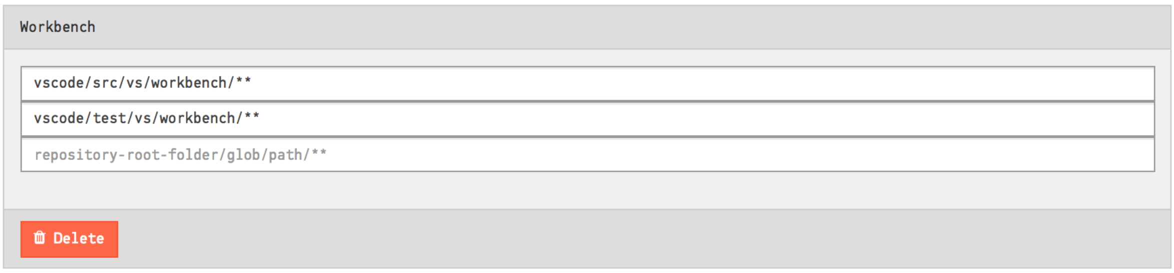
Fig. 4.81 provides an example on a CSV file used to import architectural components.



```
Workbench, vscode/src/vs/workbench/**
Workbench, vscode/test/vs/workbench/**
Code, vscode/src/vs/code/**
Editor, vscode/src/vs/editor/**
Base, vscode/src/vs/base/**
Platform, vscode/src/vs/platform/**
```

Fig. 4.81: Import your definitions of architectural component from a CSV file with this format.

The file content above defines five architectural components and maps each one of the to a logical architectural name. As you see, you can map several folders to the same architectural component. The *Workbench* component above is an example on this. As we import the file, CodeScene will generate a definition for *Workbench* as illustrated in Fig. 4.82.



```
Workbench
vscode/src/vs/workbench/**
vscode/test/vs/workbench/**
repository-root-folder/glob/path/**
```

Fig. 4.82: Map two separate folders to the same architectural component.

You can also share architectural components between projects by exporting them to a CSV file and then importing them in another project.

System Complexity Trend

CodeScene calculates a trend of how your system, as a whole, has evolved over time.

Please note that you need to enable this analysis; It's expensive in terms of analysis time, which is why it's optional. Fig. 4.83 shows how to enable the trends.

Once you've enabled the architectural trends, CodeScene will calculate an overall view of the evolution of your system as illustrated in Fig. 4.84.

You use this information to see if the system has stabilized and entered a maintenance phase or if it still evolves rapidly. You can also correlate the growth patterns to how the staffing has looked over time - did more people really resulted in a faster growth?

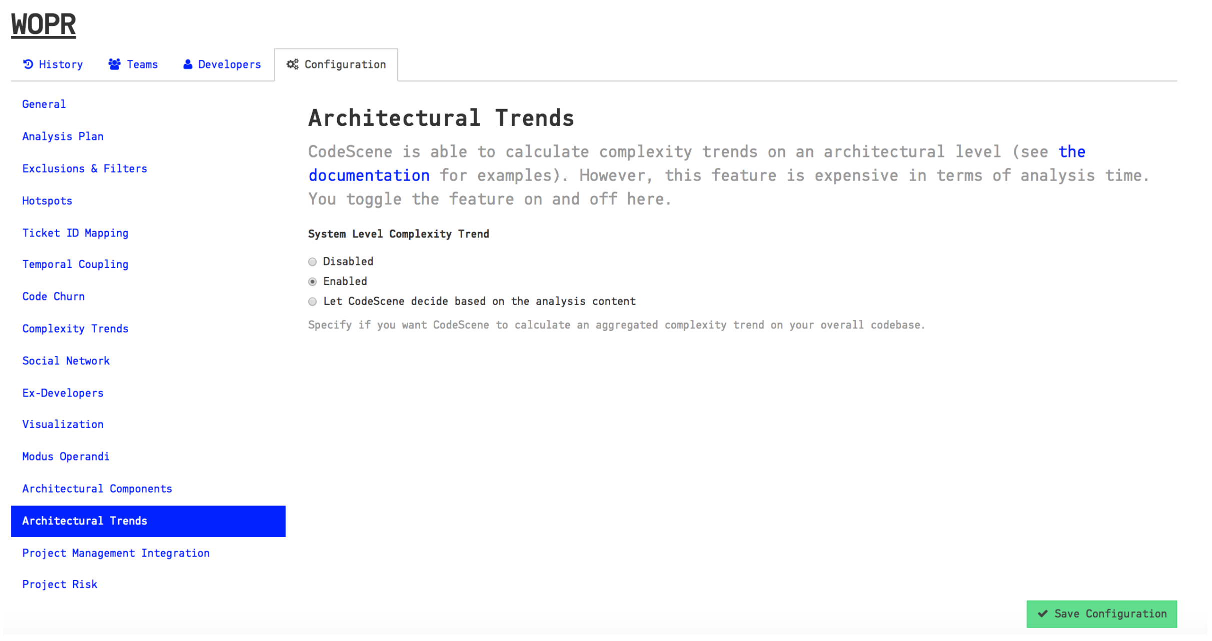
CodeScene also presents a breakdown of the system complexity per architectural component as illustrated in Fig. 4.85.

Know the Biases in System Complexity Trends

The system/architectural complexity trends don't take all your historic development into consideration. The trends are based upon the active amount of code. That is, only the code that's included in your repositories today will be considered. More specific, this means that:

- If you have deleted whole files and folders in your codebase it won't reflect in the trends.

We also want you to be careful when interpreting the results of an analysis that use a shorter time span than that of the whole repository lifetime. In such a shorter analysis period, only the files with active development activity are included in the codebase. You'll still be able to see a trend and reason about



Architectural Trends

CodeScene is able to calculate complexity trends on an architectural level (see the documentation for examples). However, this feature is expensive in terms of analysis time. You toggle the feature on and off here.

System Level Complexity Trend

- ○ Disabled
- ◉ Enabled
- ○ Let CodeScene decide based on the analysis content

Specify if you want CodeScene to calculate an aggregated complexity trend on your overall codebase.

Fig. 4.83: Enable the trend analysis of architectural components in your project configuration.

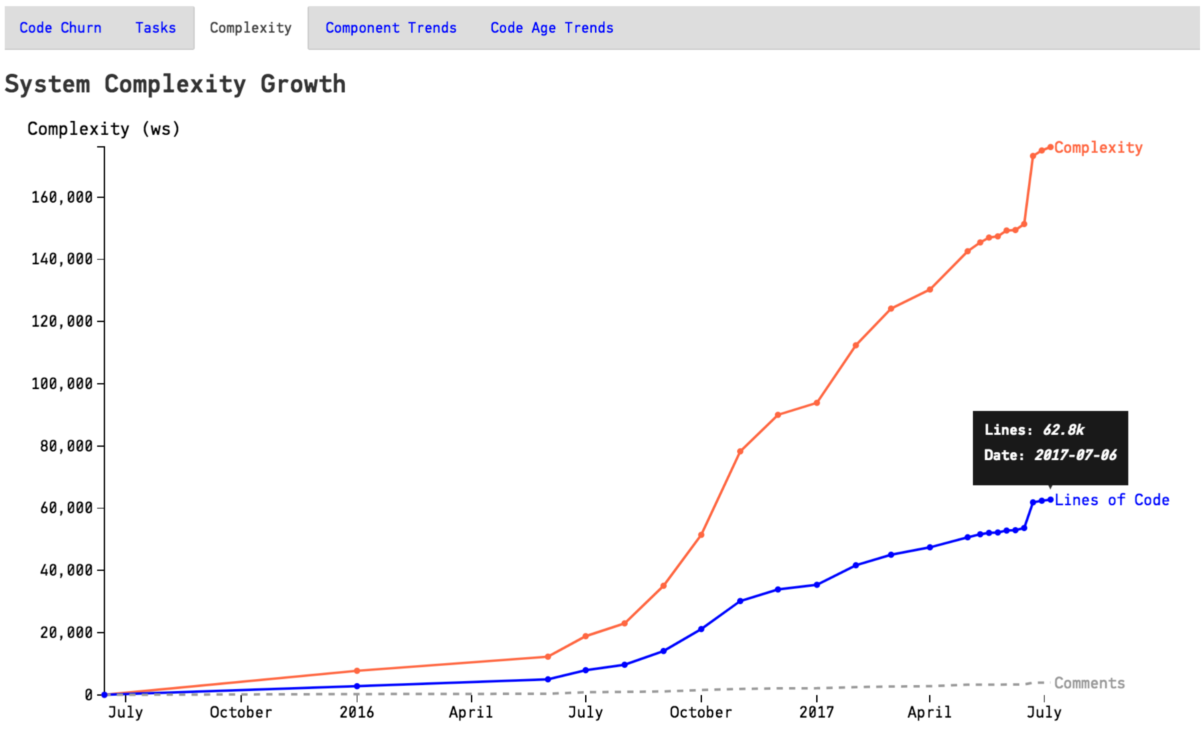


Fig. 4.84: The evolution of the complete codebase.

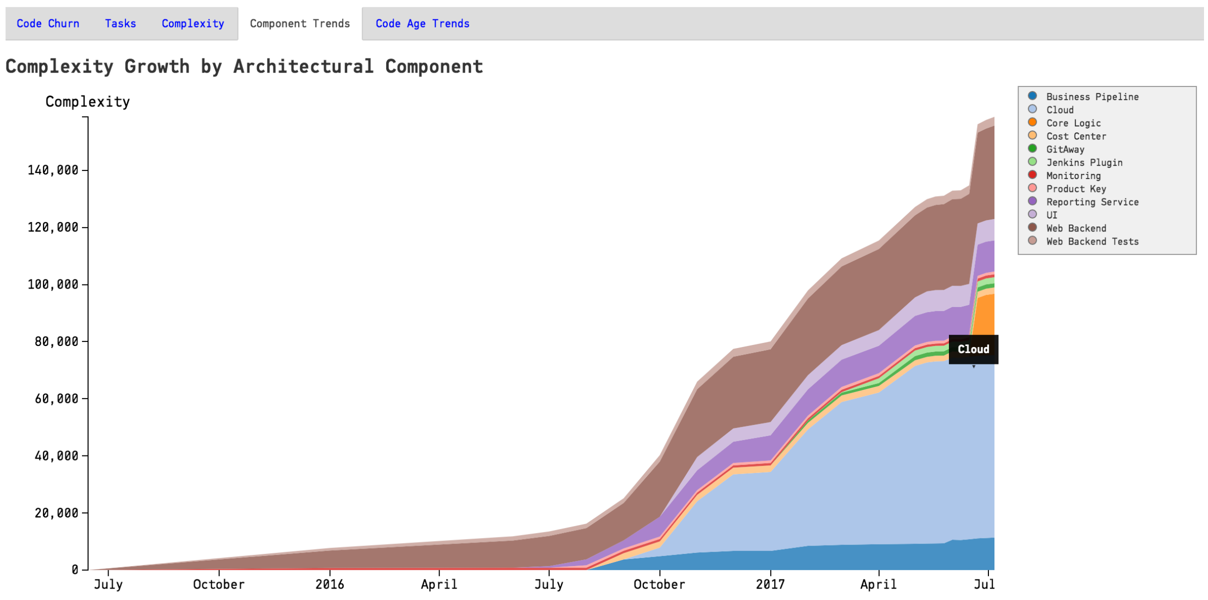


Fig. 4.85: The architectural trends let you view how the development effort has shifted over the years.

possible complexity growth in your code. However, the absolute numbers are likely to be lower than the total amount of code; Only files that you have modified are included in the trends.

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. 4.86.

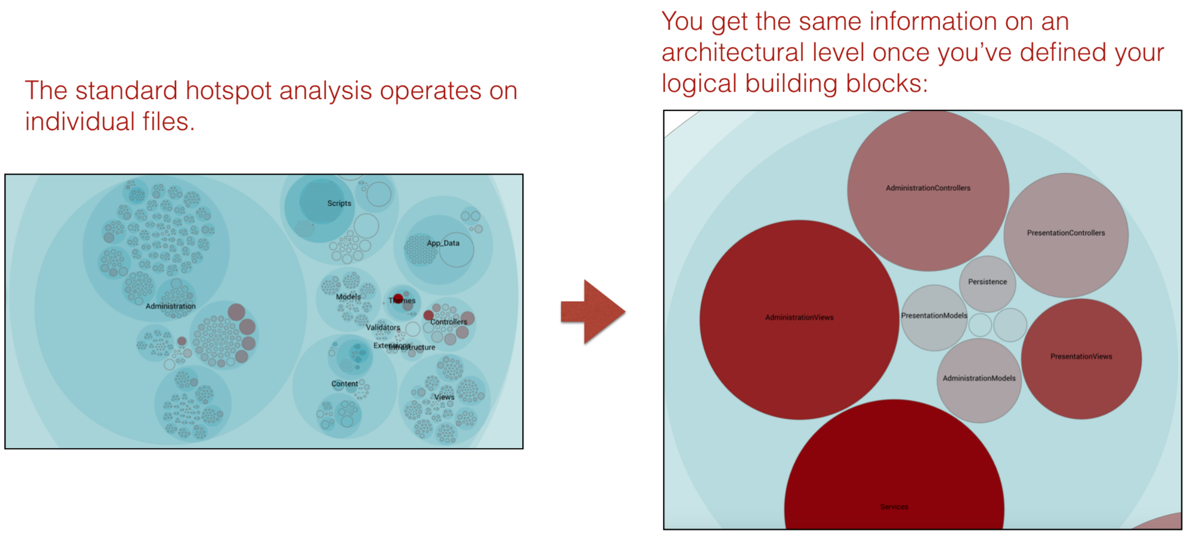


Fig. 4.86: Using the hotspot analysis for architectural components.

The architectural analyses also lets you inspect the complexity trends of architectural hotspots. Note that you need to enable the architectural trends in your project configuration as noted above.

Measure Architectural Change Coupling and Impact

The architectural analyses let you measure and visualize change coupling between architectural components.

You use this information to:

- identify expensive modification patterns,
- ensure that your dependencies match the architectural principles, and
- to measure how well your software architecture supports the way your system evolves.

The analysis also includes a trend measurement where CodeScene detects dependencies that grow stronger over time. This information is particularly useful for complex, distributed systems like microservices.

Fig. 4.87 shows an example in Spinnaker, a microservices codebase where each service is located in a separate Git repository.

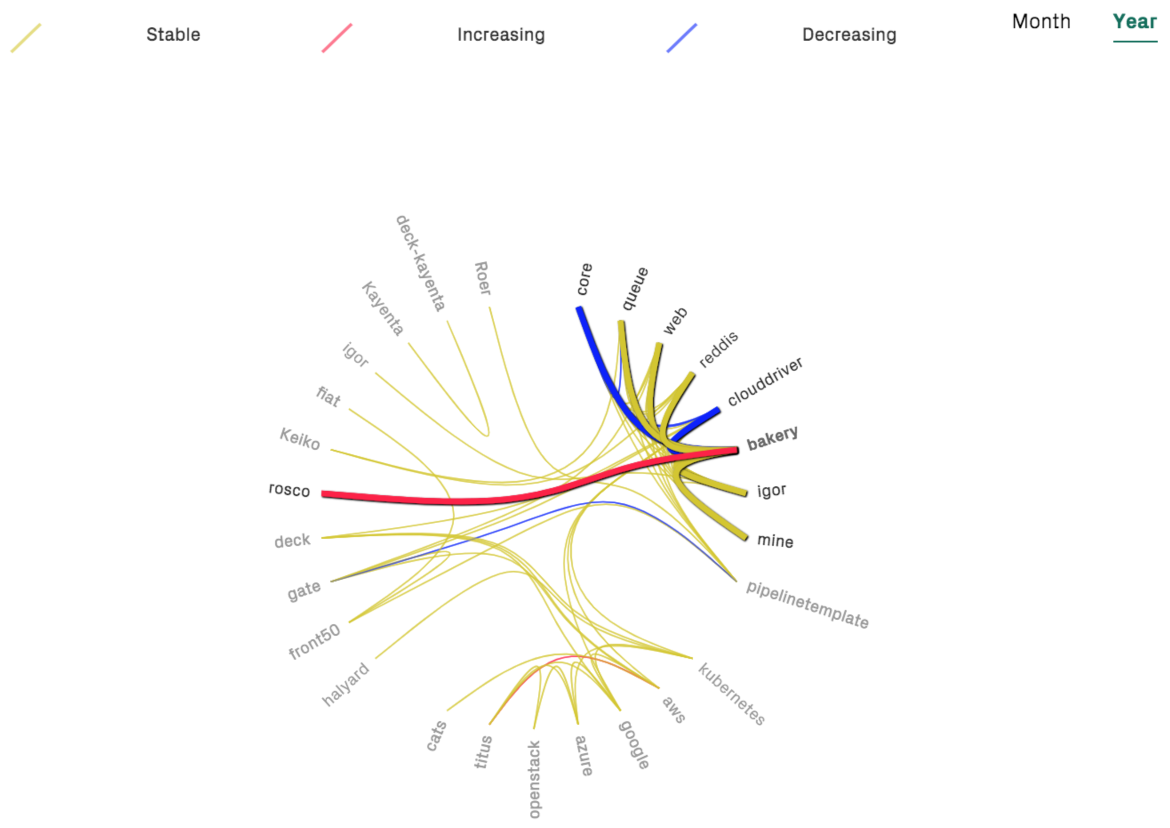


Fig. 4.87: Change coupling between different microservices.

From here, CodeScene lets you dig deeper and explore the change coupling between individual files, potentially located in separate Git repositories as shown in Fig. 4.88.

Of course, the architectural change coupling works well in a monolithic codebase as well.

Evaluate Conway's Law

CodeScene measures the knowledge distribution on an architectural level too. This gives you a powerful tool to evaluate how well your architecture aligns with your organization, aka *Conway's Law* as illustrated in Fig. 4.89.

The same analysis also lets you measure the coordination needs on an architectural level. This is useful to detect sub-systems that become coordination bottlenecks or lack a clear ownership, as illustrated in Fig. 4.90.

You use this information to find parts of the code that may have to be split into smaller parts to facilitate parallel development, or, to introduce a new team into your organization that takes on a shared responsibility.

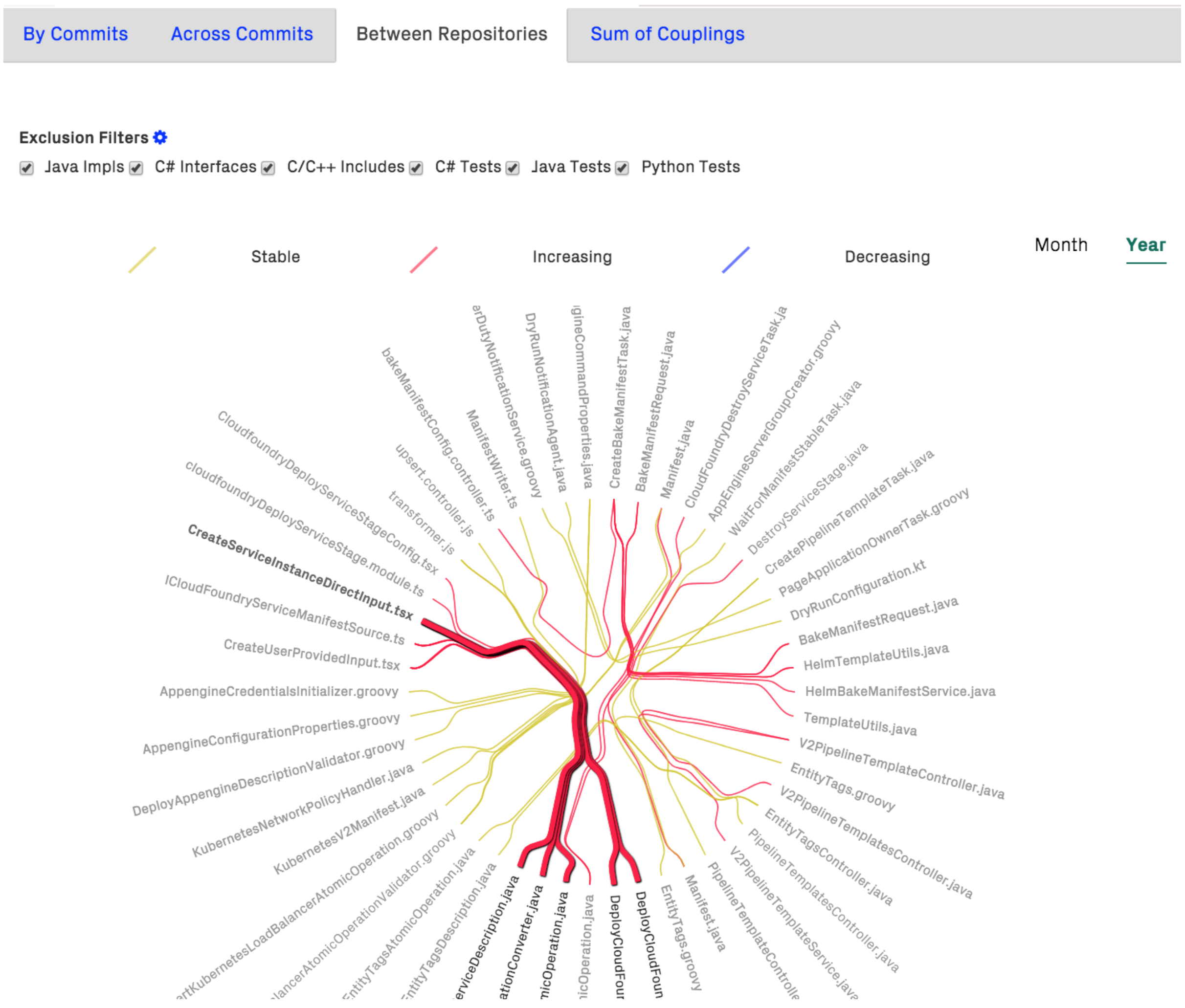


Fig. 4.88: *Change coupling between files implemented in different programming languages and located in separate Git repositories.*

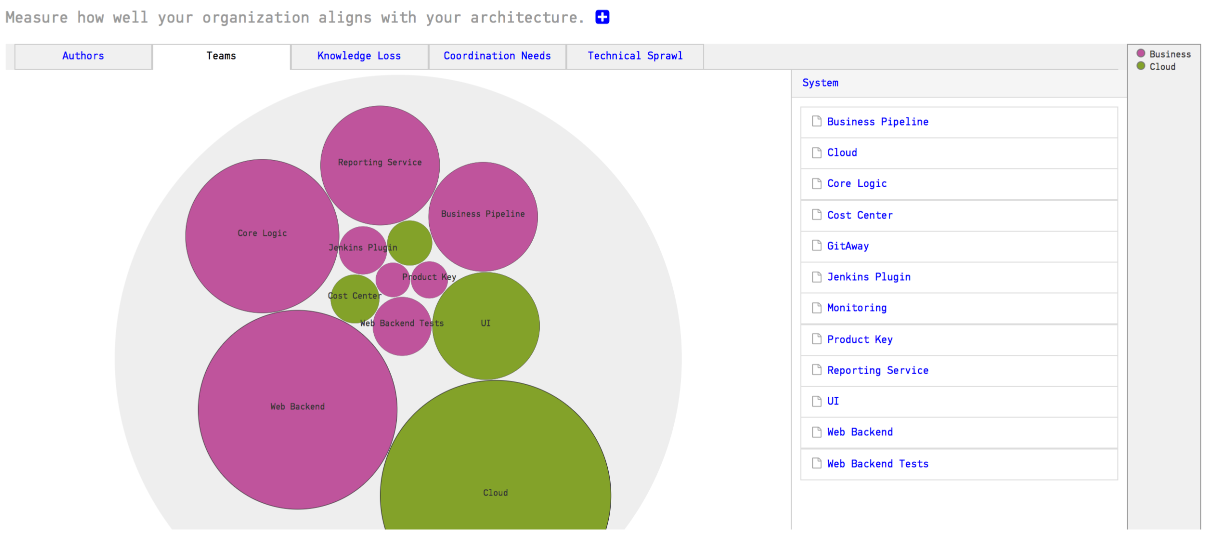
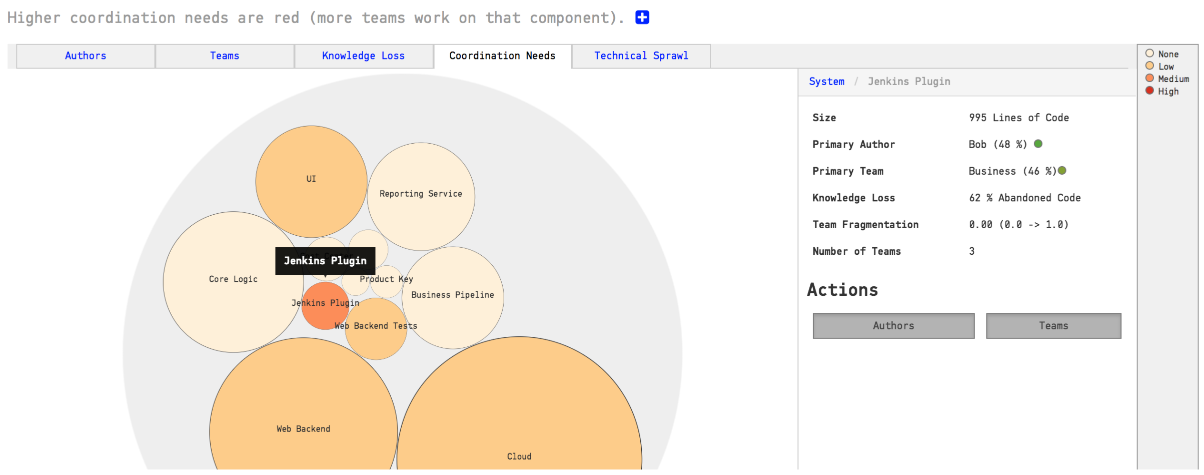


Fig. 4.89: *Measure Conway's Law in your codebase.*


Higher coordination needs are red (more teams work on that component).

System / Jenkins Plugin

Size	995 Lines of Code
Primary Author	Bob (48 %) ●
Primary Team	Business (46 %) ●
Knowledge Loss	62 % Abandoned Code
Team Fragmentation	0.00 (0.0 -> 1.0)
Number of Teams	3

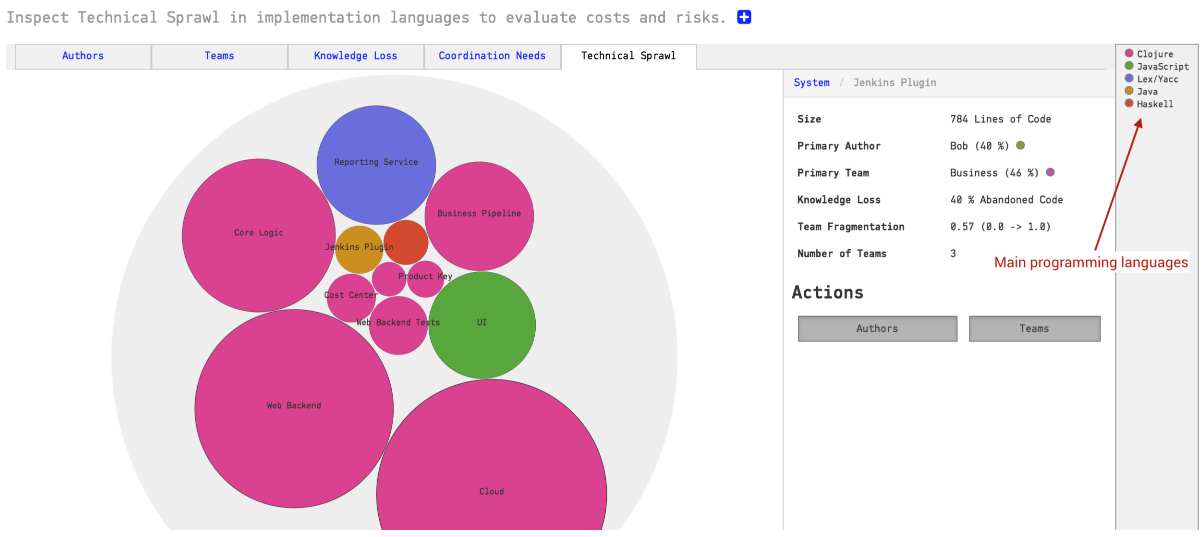
Fig. 4.90: *Find team coordination bottlenecks.*

The high-level analyses are particularly useful if you work on a (micro) service oriented architecture. In that case you also want to investigate *Technical Sprawl*, which we discuss next.

Measure Technical Sprawl

One of the big selling points behind Microservice architectures is the freedom of choice when it comes to implementation technologies. Using a Microservice architecture, each team is free to chose the programming language they think makes the best fit for the problem at hand.

In practice, however, this freedom may lead to a sprawl in programming languages that makes it hard to rotate teams. It also puts you – as an organization – at risk when the only people who master a particular technology leaves. Thus, CodeScene provides analyses to measure your technical sprawl, as illustrated in Fig. 4.91.


Inspect Technical Sprawl in implementation languages to evaluate costs and risks.

System / Jenkins Plugin

Size	784 Lines of Code
Primary Author	Bob (40 %) ●
Primary Team	Business (46 %) ●
Knowledge Loss	40 % Abandoned Code
Team Fragmentation	0.57 (0.0 -> 1.0)
Number of Teams	3

Fig. 4.91: *Technical Sprawl shows the main programming language used for each component or service.*

The technical sprawl analysis is particularly useful for off-boarding. Let's say that we want to move a developer to another project or, worse, someone decides to leave the organization. In that case we run a pro-active simulation of knowledge loss (see *Knowledge Distribution* (page 105)) and ensure that we still have the technical competencies we need within the organization, as illustrated in Fig. 4.92.

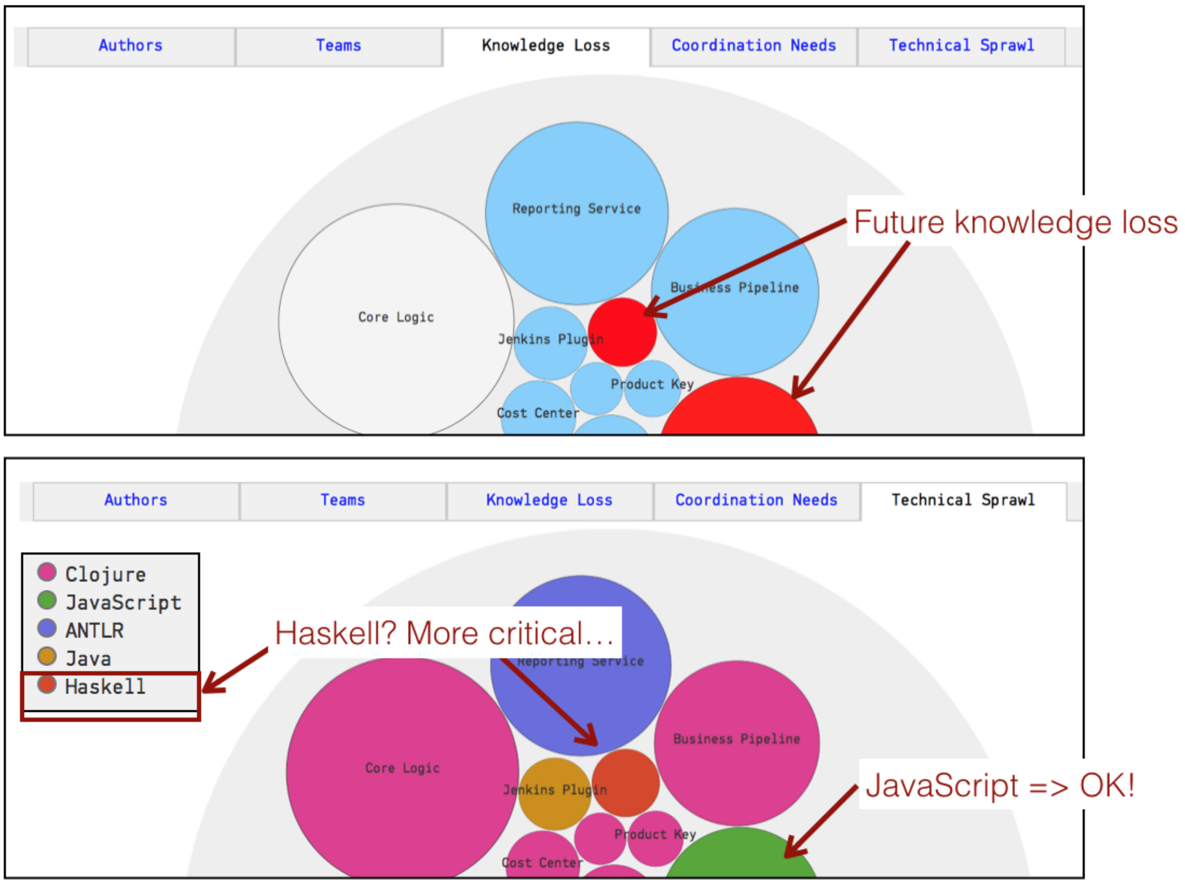


Fig. 4.92: Combine Technical Sprawl with Knowledge Loss for off-boarding.

4.4 Social

4.4.1 Delivery Effectiveness by Organizational Trends

Brooks' Law states that adding more people to a late software project makes it later. The reason for this is communication and coordination overhead. While we add more people to a project, the total number of available hours increases linearly *but* the coordination paths increase exponentially. Hence, there's a point beyond which each additional person's hours get comsumed by the increased coordination efforts. . . and then some.

Hence, use CodeScene's development output to measure the effects of any changes in staff as shown in Fig. 4.93.

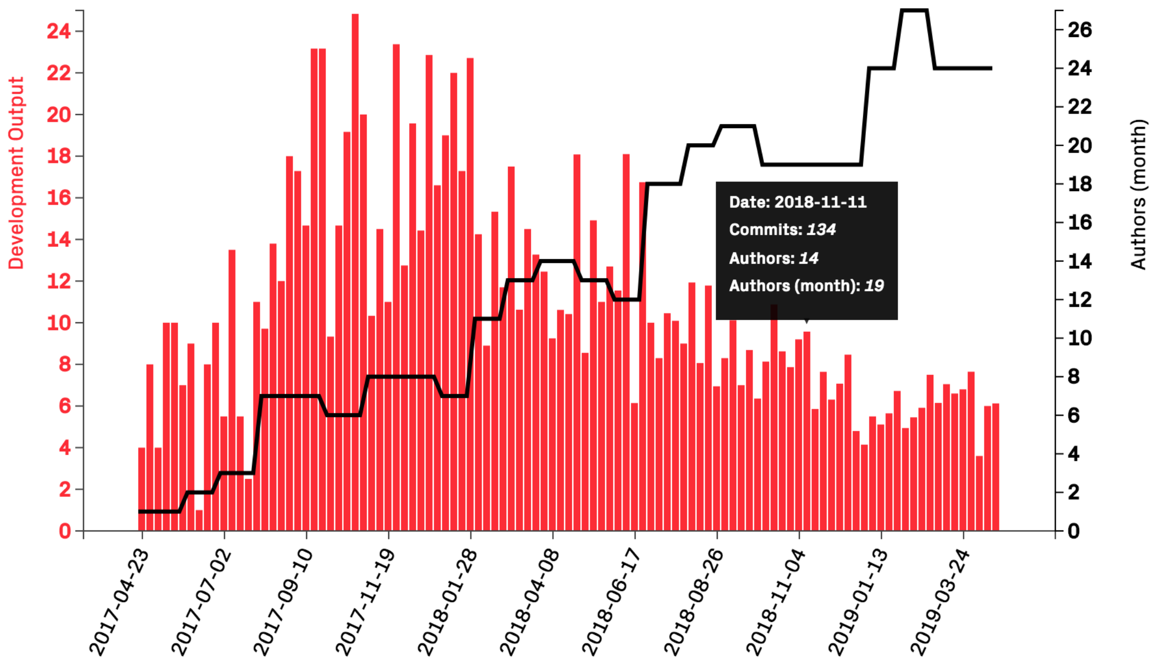


Fig. 4.93: Trends in development output with respect to the number of contributing authors.

The development output graph shows the following data:

- *Development Output*: This is measured by taking all commits during a week and dividing them by the number of contributing authors. The resulting normalized output metric gives you an estimate on the organizations's output.
- *Authors (month)*: This trend shows how many unique authors that have contributed code over a month. This trend is likely to reflect the total number of developers on the project.

If your project is at risk of falling victim to Brooks' Law, then you will see this as an increased distance between the total number of authors and their normalized output.

Like always when trying to measure things like developer and organizational productivity, the absolute numbers aren't that interesting; the interesting thing is the trend. Do you get an increase or decrease in response to changes in staffing?

Follow-up with in-depth Analyses

Should you identify a decrease in development output, then we recommend getting more information via the following analyses:

- *Coordination Needs*: Check if the teams and/or individual developers need to coordinate their work in the hotspots using the *Parallel Development and Code Fragmentation* (page 112) analysis.

- *Decrease in Code Health*: The decrease in development output could also be due to an increased level of technical debt, so inspect the *Code Biomarkers–A Virtual Code Reviewer aware of Code Health* (page 48) analysis.
- *Team Composition*: Measure the on-boarding and experience effects as described in the next section below.

Measure Team Composition Trends

While Brooks's Law is one potential reason that a project might hit a wall, there might be other explanations as well. Sometimes a development effort is appropriately staffed, but it takes time for people to get up to speed with a new codebase; on-boarding always comes with a cost. Hence, the team composition in terms of experience on your specific codebase is an important factor.

CodeScene measures team experience as shown in Fig. 4.94.

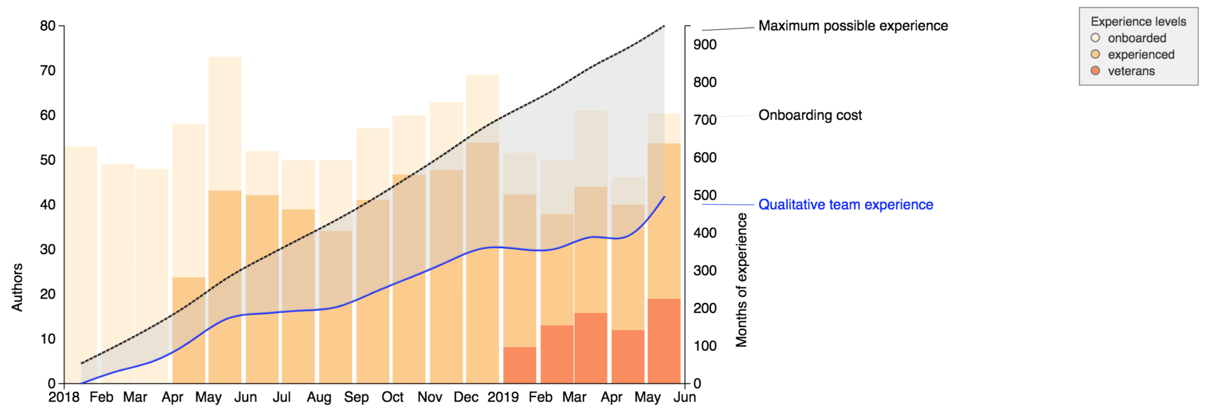


Fig. 4.94: The team composition with respect to experience accumulation and on-boarding.

The team composition visualization includes the following information based on the actual contribution span of each author:

1. Monthly team composition in terms of experience on this particular product. There are three categories: *onboarded* (0-3 months), *experienced* (6-12 months), and *veterans* (+12 months).
2. Total accumulated experience in terms on months worked on the codebase (black line).
3. "Qualitative" team experience: this is a weighted value where we consider the experience of each developer currently in the team (blue line). On-boarding a developer will come at a slight cost for a period of time. The model also takes ramp-up effects into consideration.

Often, the area between the black and blue lines is the interesting part. The wider the gap, the higher the on-boarding costs. On-boarding can of course also be viewed as an investment. From that perspective, the area between the lines shows the unrealized potential if the organization manages to keep the team stable and avoid high author churn.

Highlight the Effects of Author Churn

The team composition analysis is also useful to highlight the effects of *High Author Churn*: Taking people in and out of a project comes with a cost due to lower system mastery and repeated on-boarding effects. Keeping a team stable allows for learning and is very likely to have a positive impact on the development output.

Fig. 4.95 shows the effects on high author churn, where the qualitative team experience fails to accumulate:

When used together with the Brooks's Law development output trends, these graphs help visualizing the costs and effects over on-boarding and staffing changes over time.

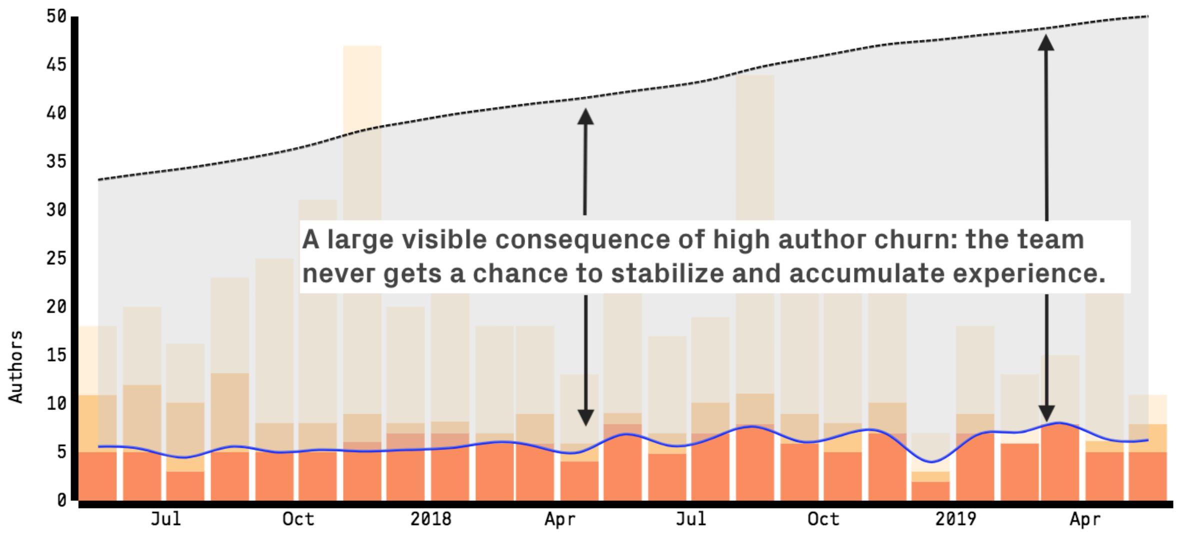


Fig. 4.95: An example on high author churn, likely to lead to low system mastery and constant on-boarding costs.

4.4.2 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 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. 4.96.

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 146)).

Define Your Development Teams

The social network lets you identify developers that should be close from an organizational perspective. The visualization in Fig. 4.96 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. 4.97.

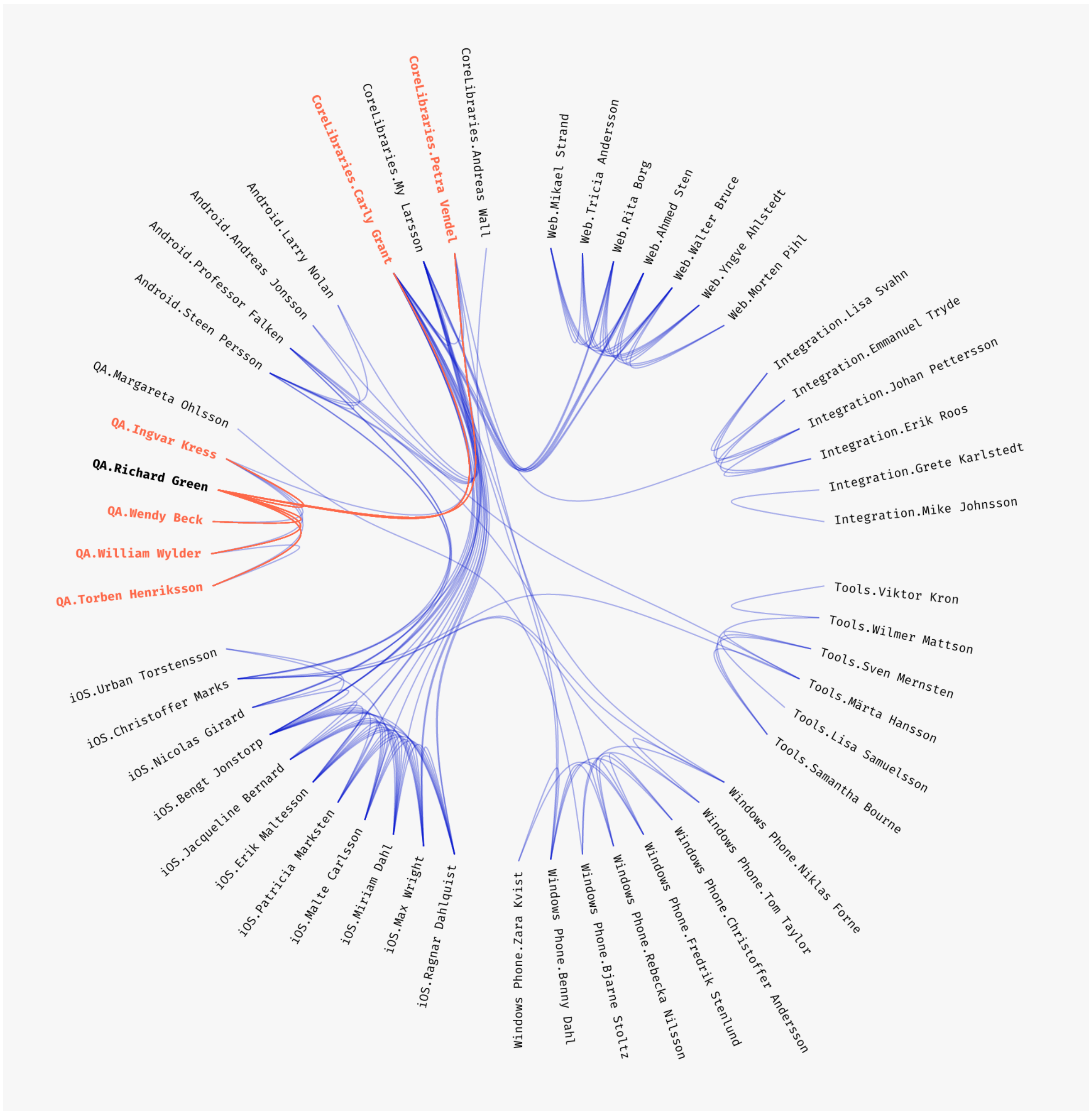


Fig. 4.96: An example of a social network in code.

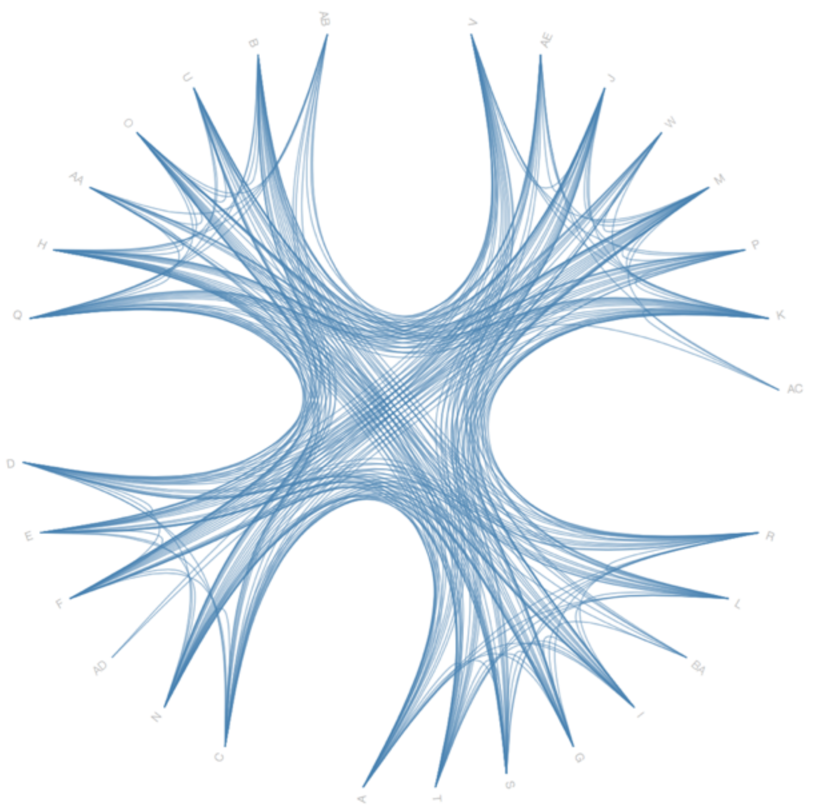


Fig. 4.97: An example of a social network anti-pattern.

The visualization in Fig. 4.97 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.

4.4.3 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 116).

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. 4.98.

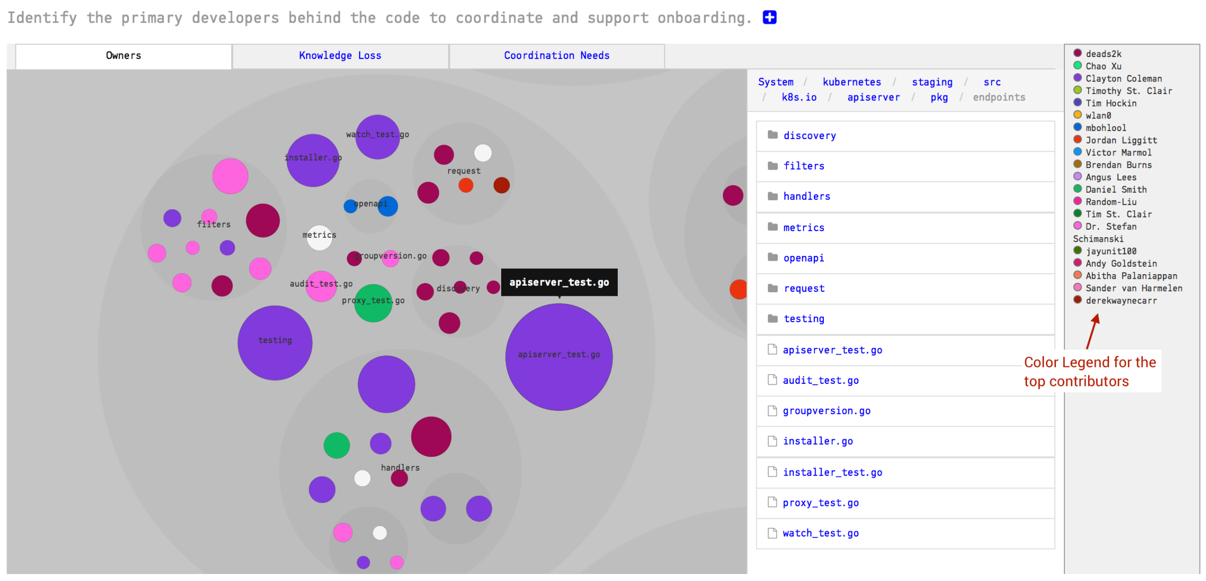


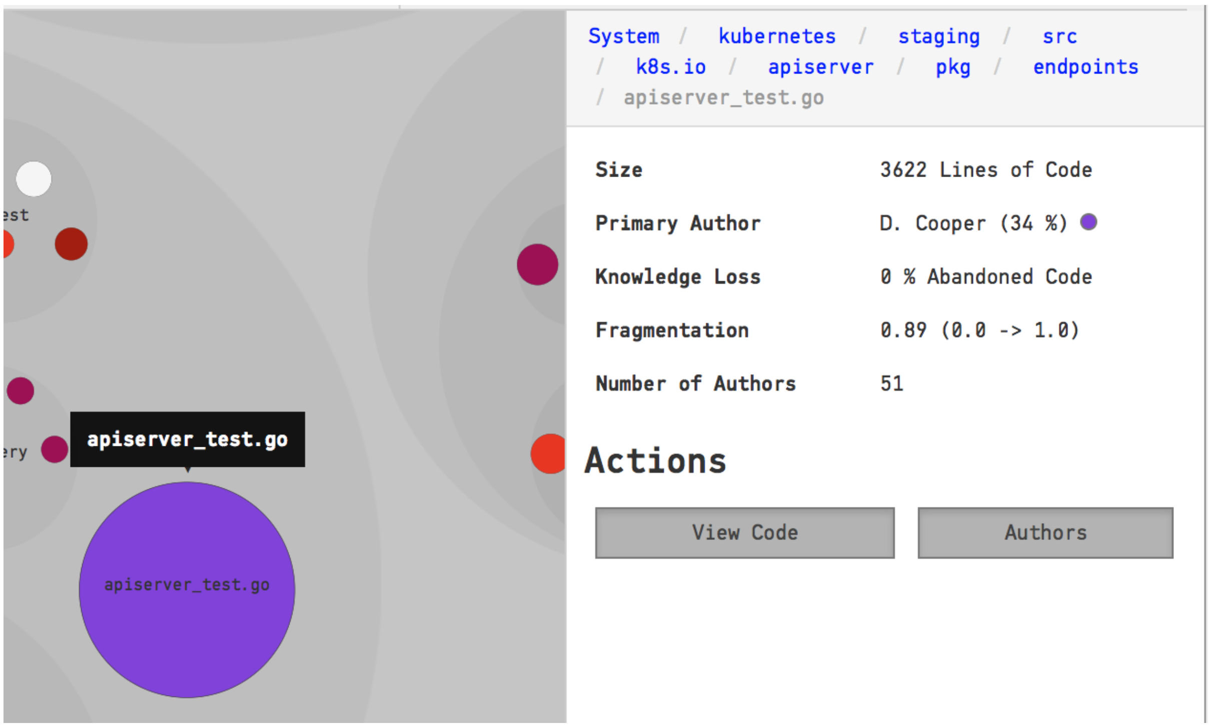
Fig. 4.98: An example of a knowledge map, click on a circle to get more information.

All knowledge maps are interactive:

- Click on a package in the visualization to zoom in on the details.
- Click outside the package to zoom out.
- Click on a circle representing a file to get detailed information.

Once you click on a file you get the option to explore who the other authors are, as shown in Fig. 4.99.

CodeScene also supports knowledge maps for pair- and mob programming, where the credits are split between the contributors in the pair. However, you need to configure your pair programming patterns in CodeScene to activate this feature. Refer to in *Configure Developers and Teams* (page 159) for the configuration options.



System / kubernetes / staging / src / k8s.io / apiserver / pkg / endpoints / apiserver_test.go

Size	3622 Lines of Code
Primary Author	D. Cooper (34 %) ●
Knowledge Loss	0 % Abandoned Code
Fragmentation	0.89 (0.0 -> 1.0)
Number of Authors	51

Actions

Fig. 4.99: Inspect the details of each file in the knowledge map.

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 159), CodeScene will accumulate their individual knowledge into their teams. The analysis results are presented using the same principles as for the Individual Knowledge Map. Only now, each color represents a team as shown in Fig. 4.100.

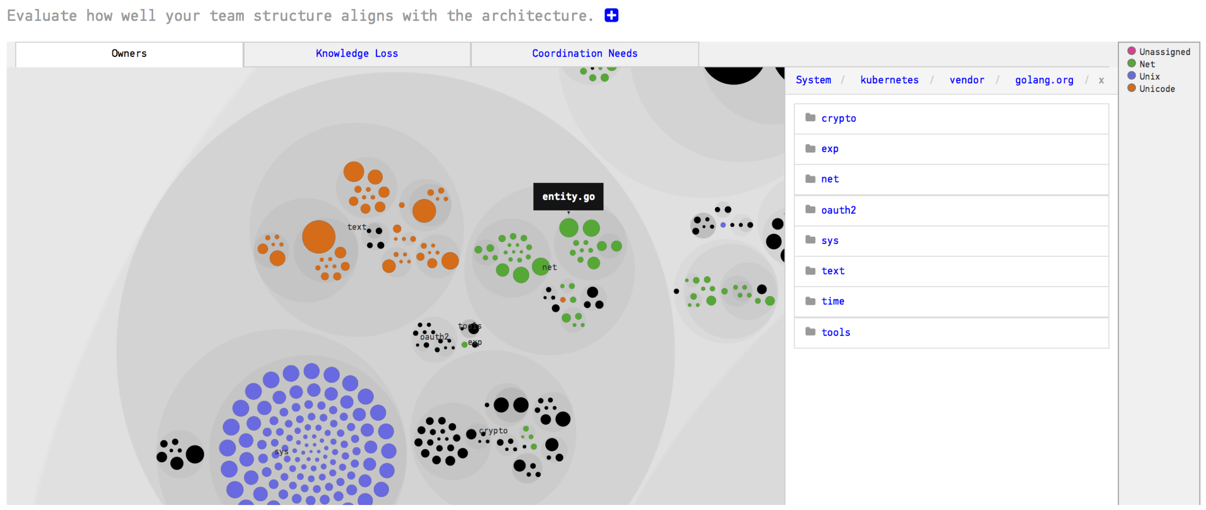


Fig. 4.100: The distribution of your teams in the codebase.

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. For example, the analysis in Fig. 4.100 has a configuration for three devlopment teams: Net, Unix, and Unicode. The analysis shows that each time has a clear area of responsibility. However, you get more details by clicking on the *Coordination Needs* aspect as shown in Fig. 4.101.

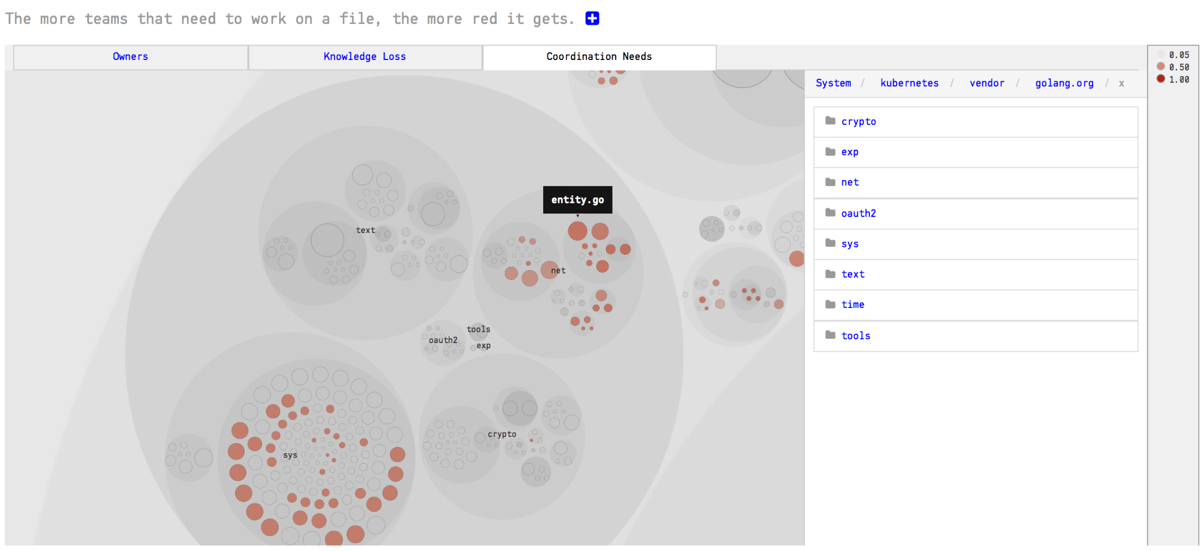


Fig. 4.101: The coordination needs between your development teams.

The coordination analysis shows you the parts of the code where multiple teams have to coordinate their work. From here you can explore which teams that are involved. The coordination analysis is also described in more detail in *Parallel Development and Code Fragmentation* (page 112).

Finally, make sure to read the discussions in the guide *Social Networks* (page 103) for more information on the organizational theories and how they correlate to the quality and efficiency of your organization.

Measure from the date of the last organizational change

Development organizations aren't static. People rotate teams, new teams are formed, and old ones abandoned. Each change introduces a possible bias into the team-level metrics.

The best way to avoid those biases is to select an analysis start date that represents the date of your last organizational change. For example, let's say you changed the team structure back in January 2017. In that case you want to start your team analysis from that date, as illustrated in Fig. 4.102.

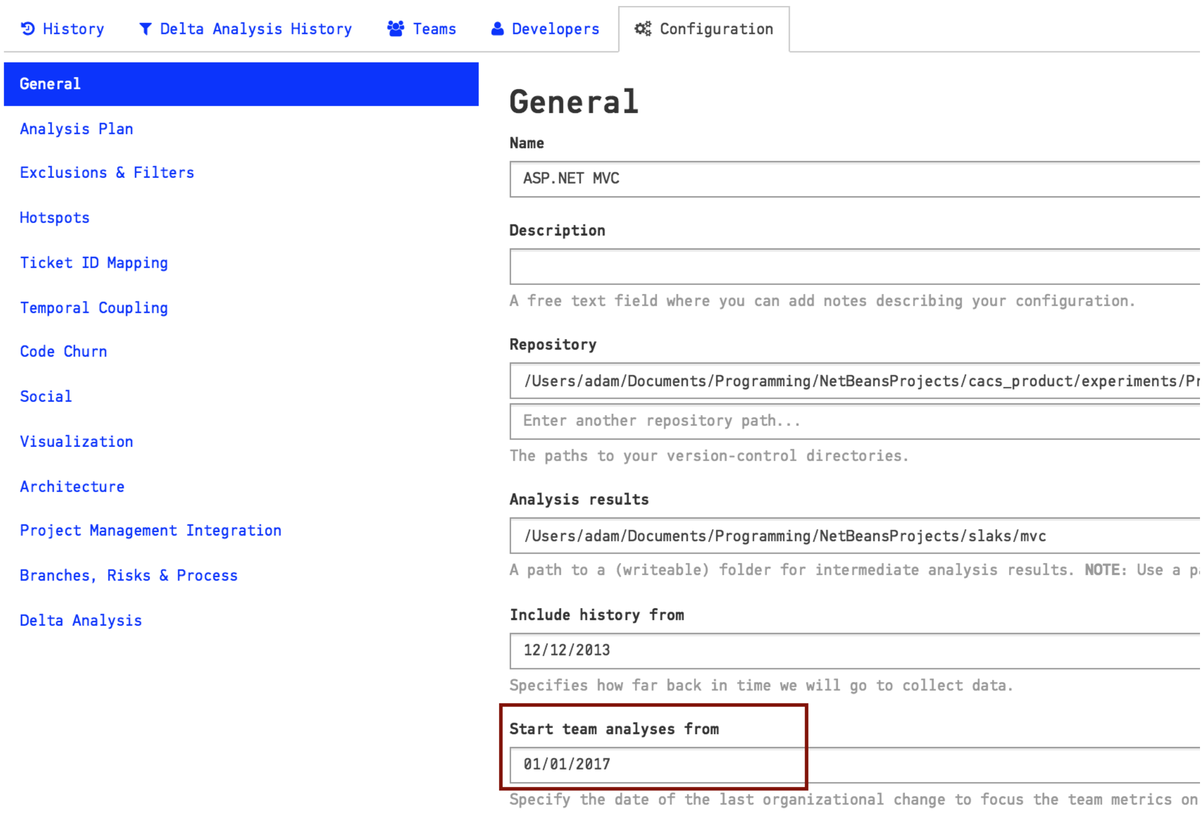
Note that you typically want to use *a longer* analysis time span for technical analyses. CodeScene resolves this by letting you configure two separate time spans, as illustrated in Fig. 4.102.

Visualize Code Ownership Patterns

Many Git hosting platforms (e.g. GitHub, GitLab, BitBucket) support the concept of *CODEOWNERS*. *CODEOWNERS* is a file where your organization can specify code owners for different parts of your codebase. The ownership is specified by using a set of glob patterns that match different modules, file types, or specific content. Here's an example:

```
# Specify the default owners in case the specific patterns
# given later won't match:
*       @TheArchitect @TheMicroManager

# The last matching pattern gets precedence, so here
# we specify the owners for invidual sub-systems:
/src/frontend    @js-owner
src/backend      @go-owner
docs/*           @TechWriter
```


History | Delta Analysis History | Teams | Developers | Configuration

General
Analysis Plan
Exclusions & Filters
Hotspots
Ticket ID Mapping
Temporal Coupling
Code Churn
Social
Visualization
Architecture
Project Management Integration
Branches, Risks & Process
Delta Analysis

General

Name
ASP.NET MVC

Description
A free text field where you can add notes describing your configuration.

Repository
/Users/adam/Documents/Programming/NetBeansProjects/cacs_product/experiments/Pr
Enter another repository path...
The paths to your version-control directories.

Analysis results
/Users/adam/Documents/Programming/NetBeansProjects/slaks/mvc
A path to a (writeable) folder for intermediate analysis results. **NOTE:** Use a p

Include history from
12/12/2013
Specifies how far back in time we will go to collect data.

Start team analyses from
01/01/2017
Specify the date of the last organizational change to focus the team metrics on

Fig. 4.102: The coordination needs between your development teams.

In this examples, the hypothetical user names *@TheArchitect*, *@js-owner*, etc. would match real people in your organization. You can of course also specify e-mail addresses instead of user handles.

If you have a *CODEOWNERS* file, CodeScene will include it in the analysis. You just need to specify that path to the file since it might vary.

In a multi-repository analysis project you might of course have multiple *CODEOWNERS* files. That's OK. Should they be in different relative locations, then you just specify all options using a semicolon separated list.

CodeScene will now mine and aggregate the code ownership as shown in Fig. 4.104.

The main use case for this information is to:

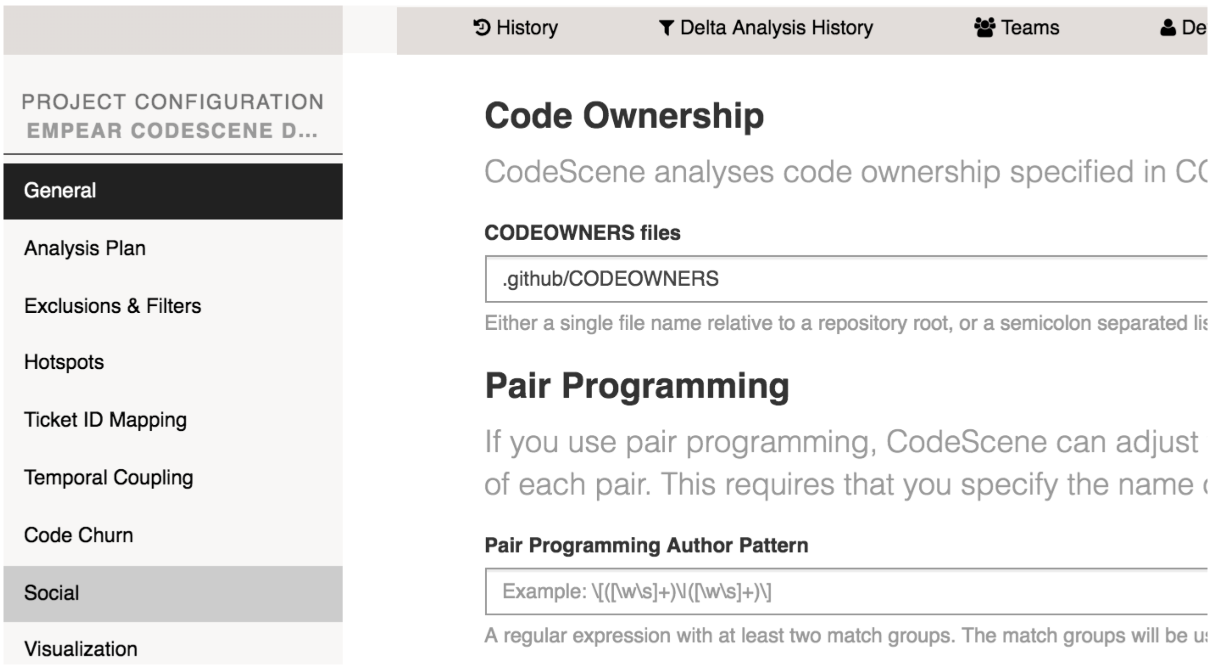
1. Ensure that no critical parts of your code lacks ownership.
2. Ensure that your hotspots have a clear and strong ownership. In particular, you want to ensure that there's a single owner for any hotspots in order to avoid diffusion of responsibility.

Notify Code Owners on Failed Quality Gates

CodeScene will also include the ownership information in case a CI/CD quality gate fails. You see an example in Fig. 4.105.

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.



Code Ownership

CodeScene analyses code ownership specified in CC

CODEOWNERS files

.github/CODEOWNERS

Either a single file name relative to a repository root, or a semicolon separated lis

Pair Programming

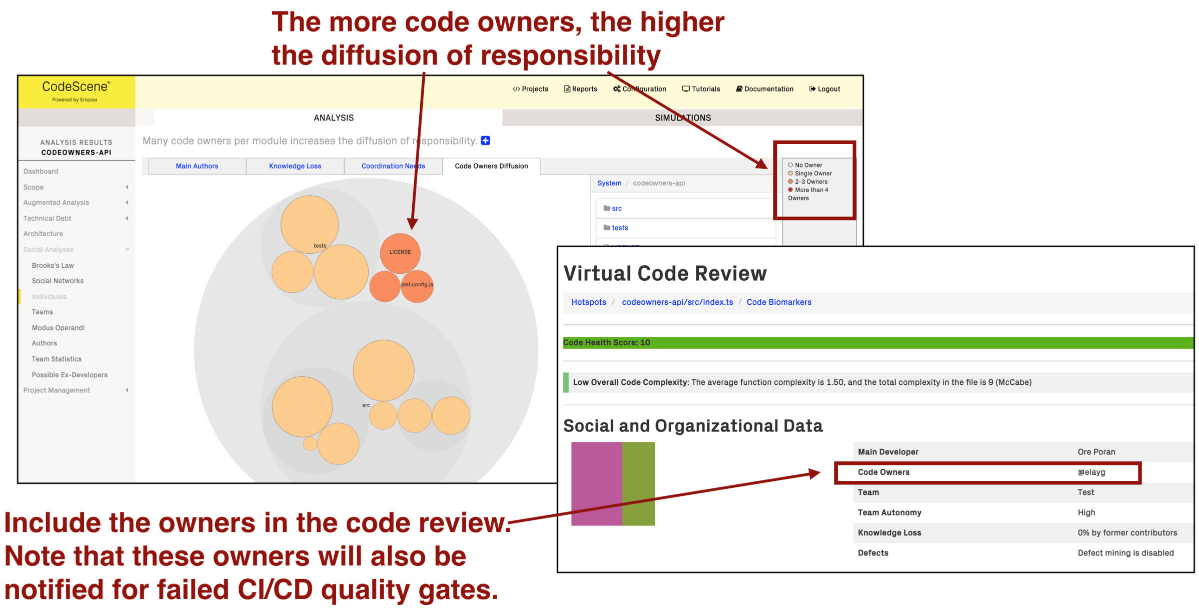
If you use pair programming, CodeScene can adjust
of each pair. This requires that you specify the name

Pair Programming Author Pattern

Example: \[([\w\s]+)\]\(([\w\s]+)\]

A regular expression with at least two match groups. The match groups will be us

Fig. 4.103: Specify the relative path to the CODEOWNERS file.



The more code owners, the higher the diffusion of responsibility

Many code owners per module increases the diffusion of responsibility.

Virtual Code Review

Code Health Score: 10

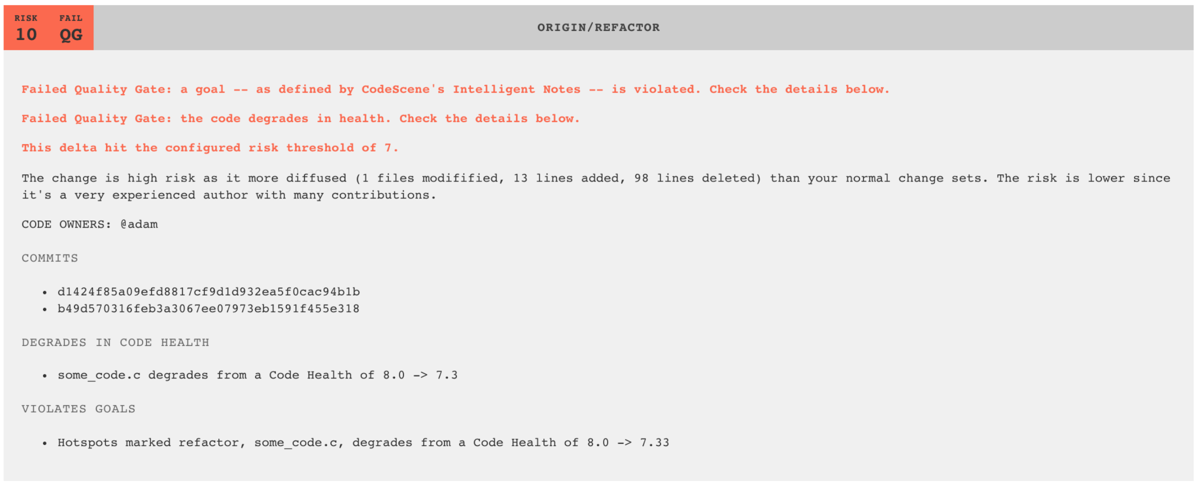
Low Overall Code Complexity: The average function complexity is 1.50, and the total complexity in the file is 9 (McCabe)

Social and Organizational Data

Main Developer	Ore Poran
Code Owners	@elayg
Team	Test
Team Autonomy	High
Knowledge Loss	0% by former contributors
Defects	Defect mining is disabled

Include the owners in the code review. Note that these owners will also be notified for failed CI/CD quality gates.

Fig. 4.104: CodeScene visualizes the code owners.



Failed Quality Gate: a goal -- as defined by CodeScene's Intelligent Notes -- is violated. Check the details below.

Failed Quality Gate: the code degrades in health. Check the details below.

This delta hit the configured risk threshold of 7.

The change is high risk as it more diffused (1 files modified, 13 lines added, 98 lines deleted) than your normal change sets. The risk is lower since it's a very experienced author with many contributions.

CODE OWNERS: @adam

COMMITS

- d1424f85a09efd8817cf9d1d932ea5f0cac94b1b
- b49d570316feb3a3067ee07973eb1591f455e318

DEGRADES IN CODE HEALTH

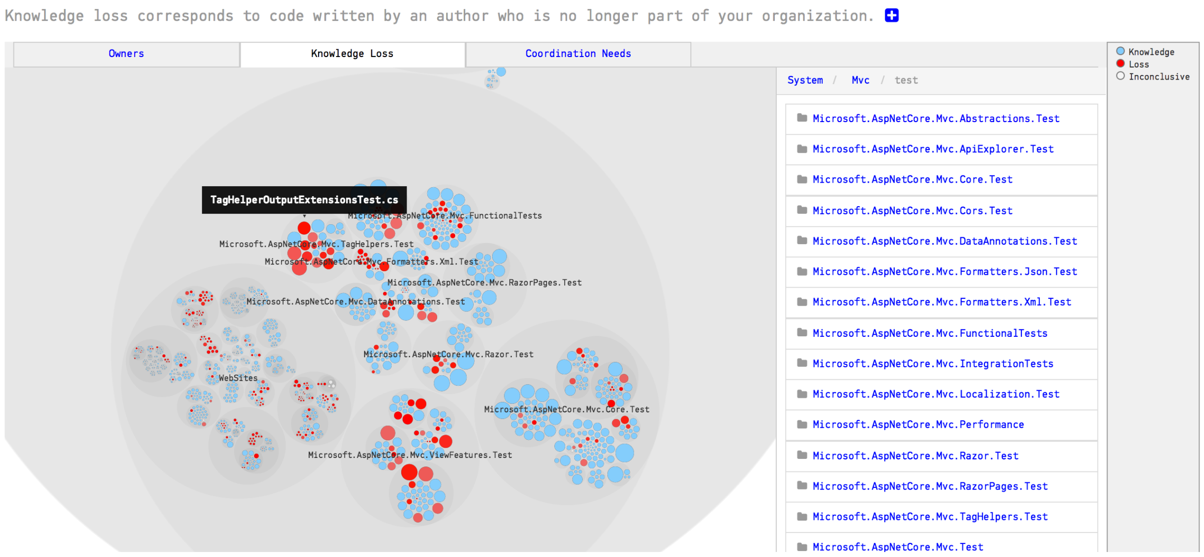
- some_code.c degrades from a Code Health of 8.0 -> 7.3

VIOLATES GOALS

- Hotspots marked refactor, some_code.c, degrades from a Code Health of 8.0 -> 7.33

Fig. 4.105: CodeScene notifies code owners when a quality gate fails.

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 159)). Those parts of the codebase that are dominated by Ex-Developers are marked as red in the knowledge loss visualization. Fig. 4.106 shows an example from an organization where some core developers have left.


Knowledge loss corresponds to code written by an author who is no longer part of your organization.

Owners
Knowledge Loss
Coordination Needs
Knowledge
Loss
Inconclusive
System / Mvc / test
Microsoft.AspNetCore.Mvc.Abstractions.Test
Microsoft.AspNetCore.Mvc.ApiExplorer.Test
Microsoft.AspNetCore.Mvc.Core.Test
Microsoft.AspNetCore.Mvc.Cors.Test
Microsoft.AspNetCore.Mvc.DataAnnotations.Test
Microsoft.AspNetCore.Mvc.Formatters.Json.Test
Microsoft.AspNetCore.Mvc.Formatters.Xml.Test
Microsoft.AspNetCore.Mvc.FunctionalTests
Microsoft.AspNetCore.Mvc.IntegrationTests
Microsoft.AspNetCore.Mvc.Localization.Test
Microsoft.AspNetCore.Mvc.Performance
Microsoft.AspNetCore.Mvc.Razor.Test
Microsoft.AspNetCore.Mvc.RazorPages.Test
Microsoft.AspNetCore.Mvc.TagHelpers.Test
Microsoft.AspNetCore.Mvc.Test
TagHelperOutputExtensionsTest.cs
Microsoft.AspNetCore.Mvc.FunctionalTests
Microsoft.AspNetCore.Mvc.TagHelpers.Test
Microsoft.AspNetCore.Mvc.Formatters.Xml.Test
Microsoft.AspNetCore.Mvc.RazorPages.Test
Microsoft.AspNetCore.Mvc.DataAnnotations.Test
Microsoft.AspNetCore.Mvc.Razor.Test
WebSites
Microsoft.AspNetCore.Mvc.Core.Test
Microsoft.AspNetCore.Mvc.ViewFeatures.Test

Fig. 4.106: An example on a knowledge loss analysis.

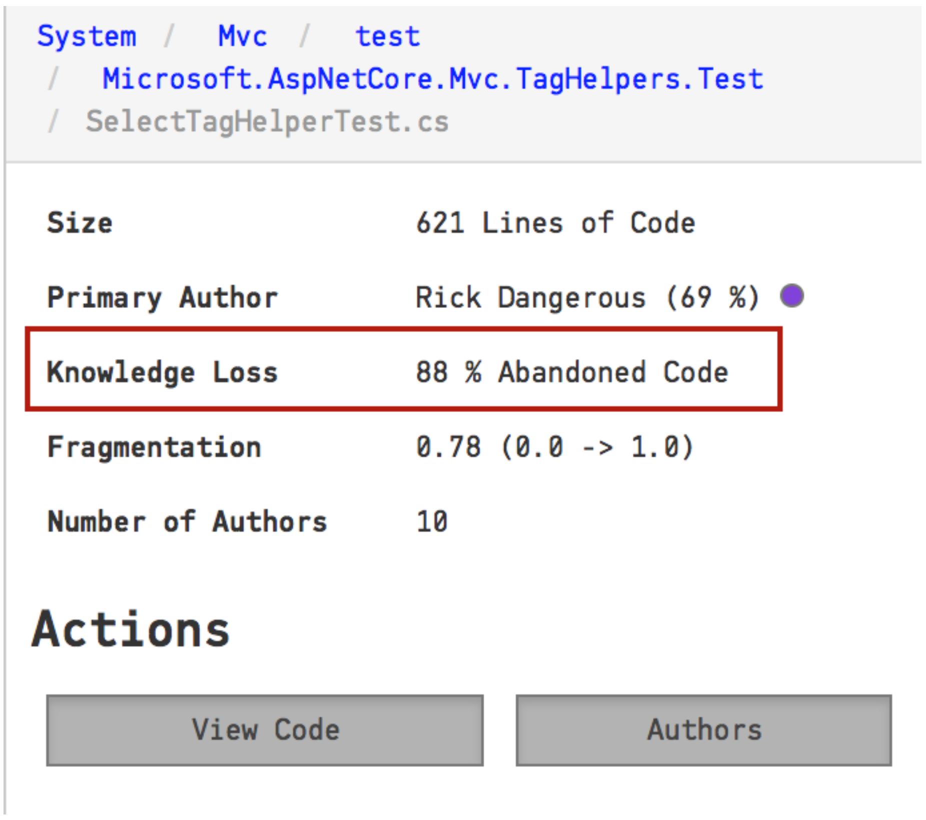
To inspect the knowledge loss you just click on a file, as shown in Fig. 4.107.

Note that there's a special label in the knowledge visualization: *Inconclusive*. Inconclusive means that CodeScene cannot determine the original author of a piece of code. This is something that happens if you run a knowledge analysis on a shorter time span than the total lifetime of a codebase. CodeScene tracks moved and renamed content, but in doing so it depends on the underlaying object model of Git. So in the rare cases where copied content doesn't get detected as such, the code may show up as inconclusive.

Use knowledge loss as a simulation

There are several uses for the knowledge loss information. In retrospect, you use it as part of your planning and risk management since it is an increased risk to modify code we no longer understand.

However, the knowledge loss analysis is much more powerful when used as a simulation. In this case you use CodeScene to simulate different scenarios and how they would affect your organization. Used this


System / Mvc / test / Microsoft.AspNetCore.Mvc.TagHelpers.Test / SelectTagHelperTest.cs

Size	621 Lines of Code
Primary Author	Rick Dangerous (69 %) ●
Knowledge Loss	88 % Abandoned Code
Fragmentation	0.78 (0.0 -> 1.0)
Number of Authors	10

Actions

Fig. 4.107: Inspect the detailed knowledge loss of a file.

way, the knowledge loss analysis becomes a pro-active tool that helps you avoid unpleasant surprises in case a contractor leaves or a developer gets moved to a different project.

The guide in *Team Planning with the On- and Off-Boarding Simulation* (page 137) describes how to simulate upcoming knowledge loss so that you can act on time.

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

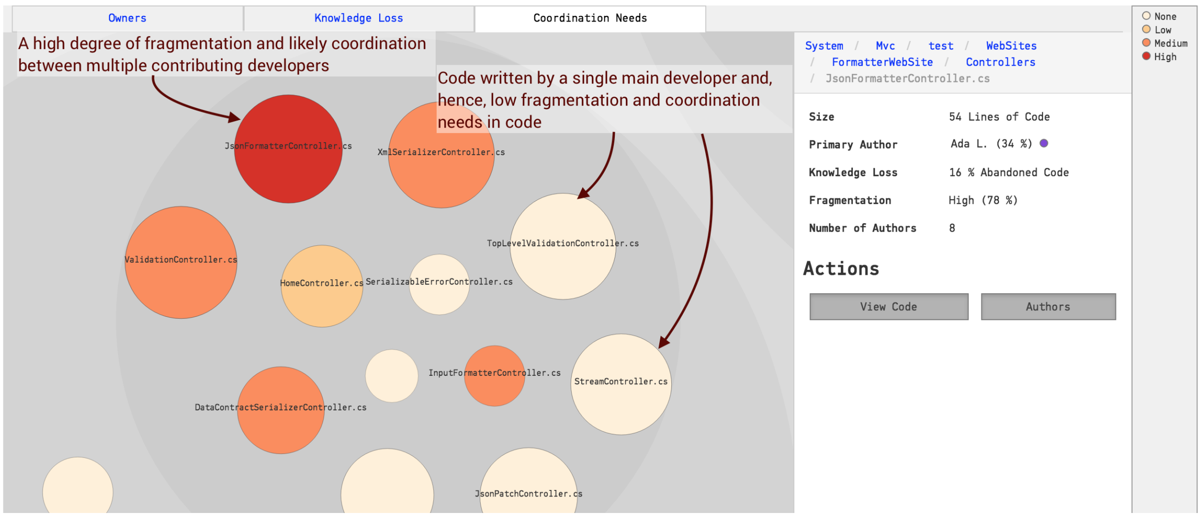
Before you read on, please note that CodeScene uses the name of each committer to calculate the fragmentation metrics. So *make sure* you understand the possible biases discussed in the guide *Know the possible Biases in the Data* (page 116).

The Coordination Needs 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 your organization doesn't have any team structure, start with investigating the fragmentation by authors as illustrated in Fig. 4.108.



Owners
Knowledge Loss
Coordination Needs
A high degree of fragmentation and likely coordination between multiple contributing developers
Code written by a single main developer and, hence, low fragmentation and coordination needs in code
JsonFormatterController.cs
XmlSerializerController.cs
TopLevelValidationController.cs
ValidationController.cs
HomeController.cs
SerializableErrorController.cs
InputFormatterController.cs
StreamController.cs
DataContractSerializerController.cs
JsonPatchController.cs
System / Mvc / test / WebSites / FormatterWebSite / Controllers / JsonFormatterController.cs
Size 54 Lines of Code
Primary Author Ada L. (34 %)
Knowledge Loss 16 % Abandoned Code
Fragmentation High (78 %)
Number of Authors 8
Actions
View Code
Authors
None
Low
Medium
High

Fig. 4.108: The fragmentation map shows files with excess parallel development.

The fragmentation map in Fig. 4.108 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 100%: The closer to 100% 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.

Get more Detailed Information with Fractal Figures

The fragmentation map in Fig. 4.108 is interactive. That means you can click on each file and inspect the amount of fragmentation as illustrated in Fig. 4.109.

4.4.5 Modus Operandi

Modus Operandi is the method of operation. It's the signature for how you work with the codebase and lets you discover trends 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 118)) 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.

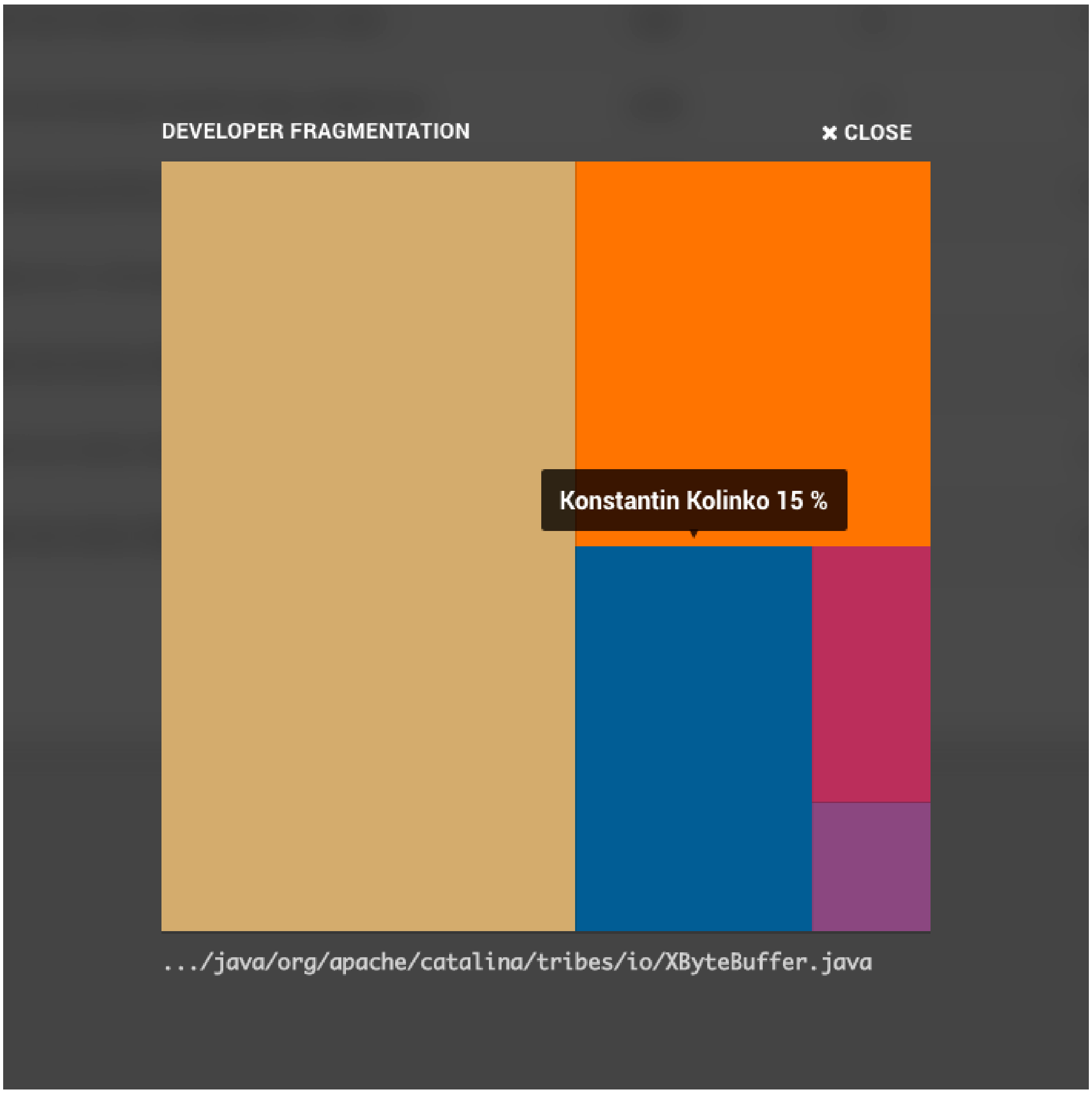
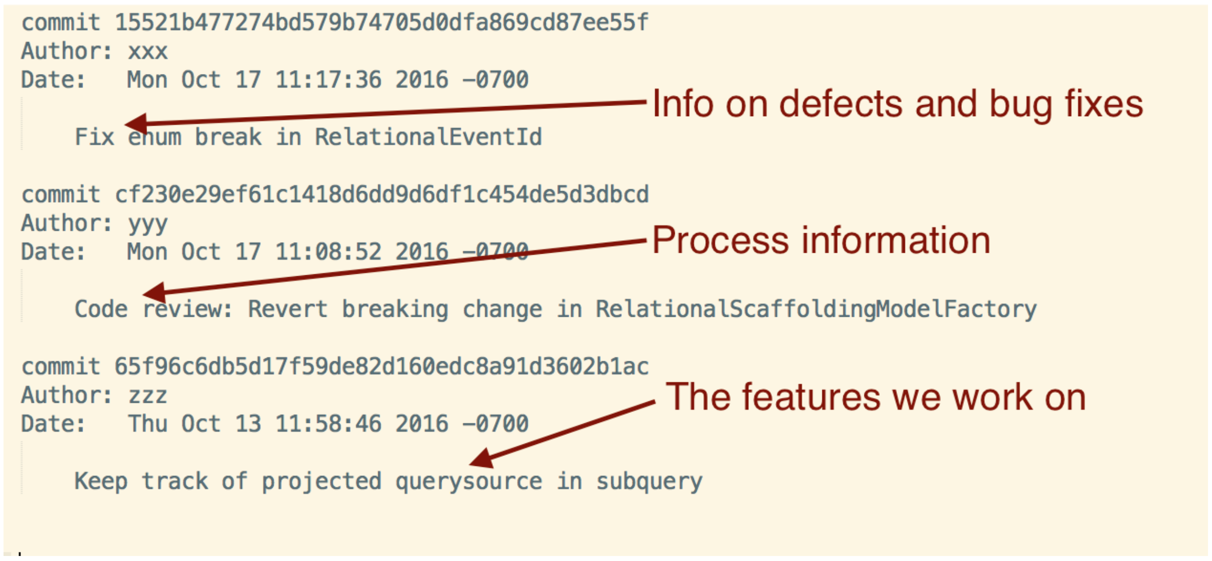


Fig. 4.109: An example of a developer fragmentation. Hovering a colored fragment shows the developer and the relative contribution.

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. 4.110



```
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. 4.110: Your commit messages contains work-related information.

By default, CodeScene will identify all commit messages that contain the texts *bug*, *fix*, or *defect*, but you can override the defaults and provide any patterns as illustrated in Fig. 4.111.

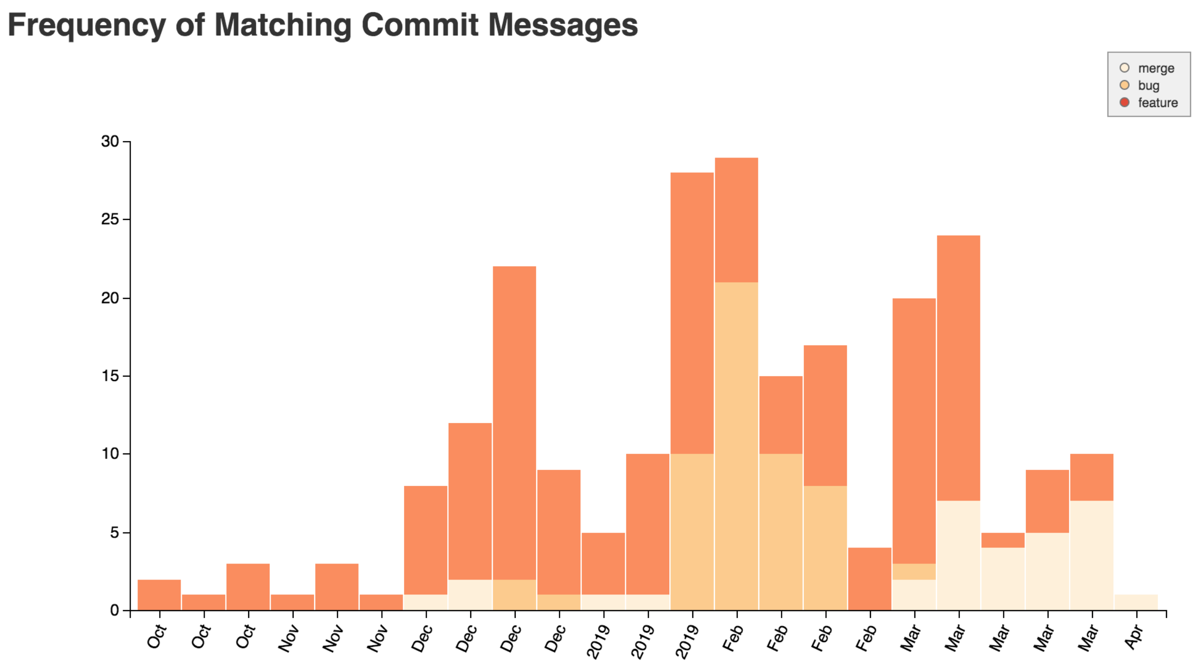


Fig. 4.111: Inspect all commits that mention a particular word or phrase.

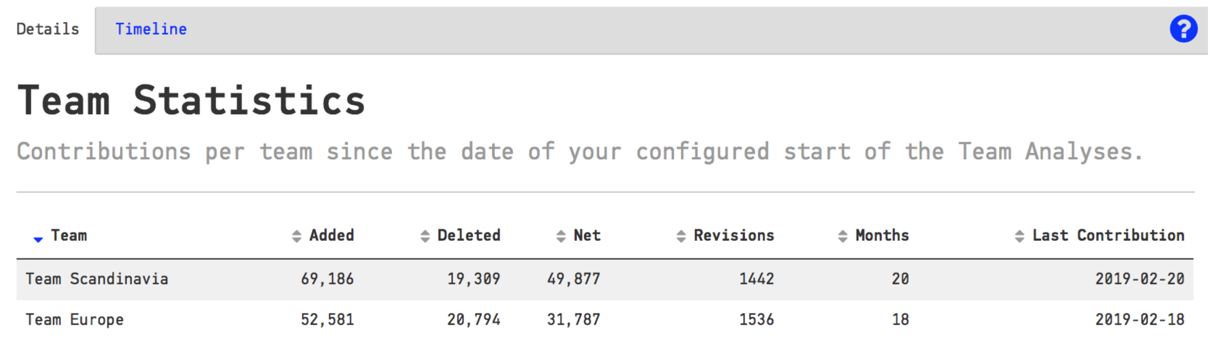
Note that the matches are always case insensitive. That is, if you specify *bug*, CodeScene will match both *bug* and *Bug*.

4.4.6 Author and Team Statistics

CodeScene provides an aggregated view of all author and team contributions.

Detailed Team Statistics

The team statistics let you view the contributions aggregated on a team level as show in Fig. 4.112.



Team Statistics

Contributions per team since the date of your configured start of the Team Analyses.

Team	Added	Deleted	Net	Revisions	Months	Last Contribution
Team Scandinavia	69,186	19,309	49,877	1442	20	2019-02-20
Team Europe	52,581	20,794	31,787	1536	18	2019-02-18

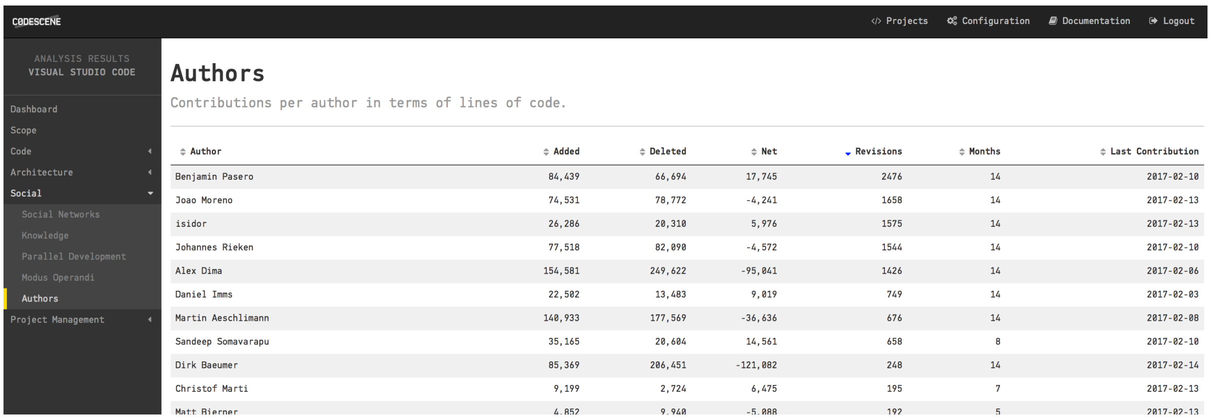
Fig. 4.112: The detailed team statistics show the aggregated contributions on a team level.

The detailed team statistics are calculated based on the analysis start dates for team analyses.

The team statistics also include a timeline that shows how the contributions – on a team level – are distributed over time (see the author example below for an example – the team view provides the same data).

Detailed Author Statistics

The author information is intended as descriptive data that lets you find long-term contributors as shown in Fig. 4.113.



Authors

Contributions per author in terms of lines of code.

Author	Added	Deleted	Net	Revisions	Months	Last Contribution
Benjamin Pasero	84,439	66,694	17,745	2476	14	2017-02-10
Joao Moreno	74,531	78,772	-4,241	1658	14	2017-02-13
isidor	26,286	20,310	5,976	1575	14	2017-02-13
Johannes Rieken	77,518	82,090	-4,572	1544	14	2017-02-10
Alex Dima	154,581	249,622	-95,041	1426	14	2017-02-06
Daniel Imms	22,502	13,483	9,019	749	14	2017-02-03
Martin Aeschlimann	140,933	177,569	-36,636	676	14	2017-02-08
Sandeep Somavarapu	35,165	20,604	14,561	658	8	2017-02-10
Dirk Baeumer	85,369	206,451	-121,082	248	14	2017-02-14
Christof Marti	9,199	2,724	6,475	195	7	2017-02-13
Matt Bierner	4,852	9,940	-5,088	192	5	2017-02-13

Fig. 4.113: The detailed author statistics show the aggregated contributions.

CodeScene also calculates a timeline with a heatmap of the ontributions by each active author, as shown in Fig. 4.114.

The author heatmap shows the number of recorded commits by each active author. Note that the contributions are filtered according to your project configuration. That is, author contributions to blacklisted or excluded content aren't included in the statistics.

4.4.7 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.

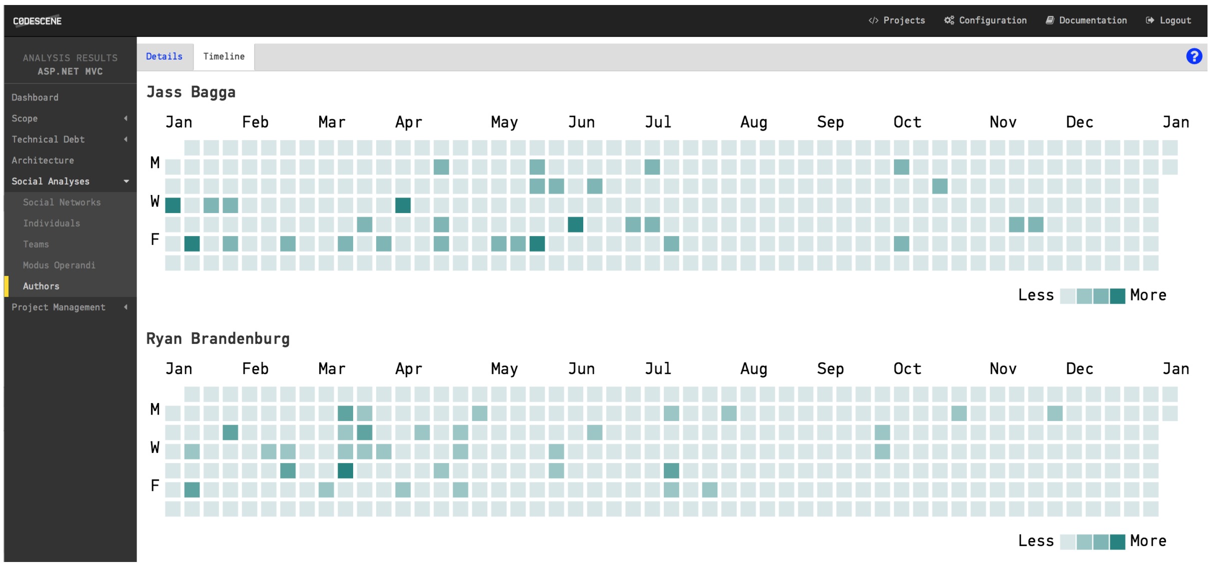


Fig. 4.114: The timeline shows a heatmap of active author contributions.

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, you can avoid this bias by resolving the author aliases in CodeScene's configuration UI. As an alternative to the UI, you may also use 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.

CodeScene does supports knowledge maps for pair and mob programming, where the credits are split between the contributors in the pair. Refer to in *Configure Developers and Teams* (page 159) for the configuration options needed to activate this feature.

4.5 Project Management

4.5.1 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 analysis lets you assess costs on both the architectural level, such as components and sub-systems, as well as on individual files.

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?

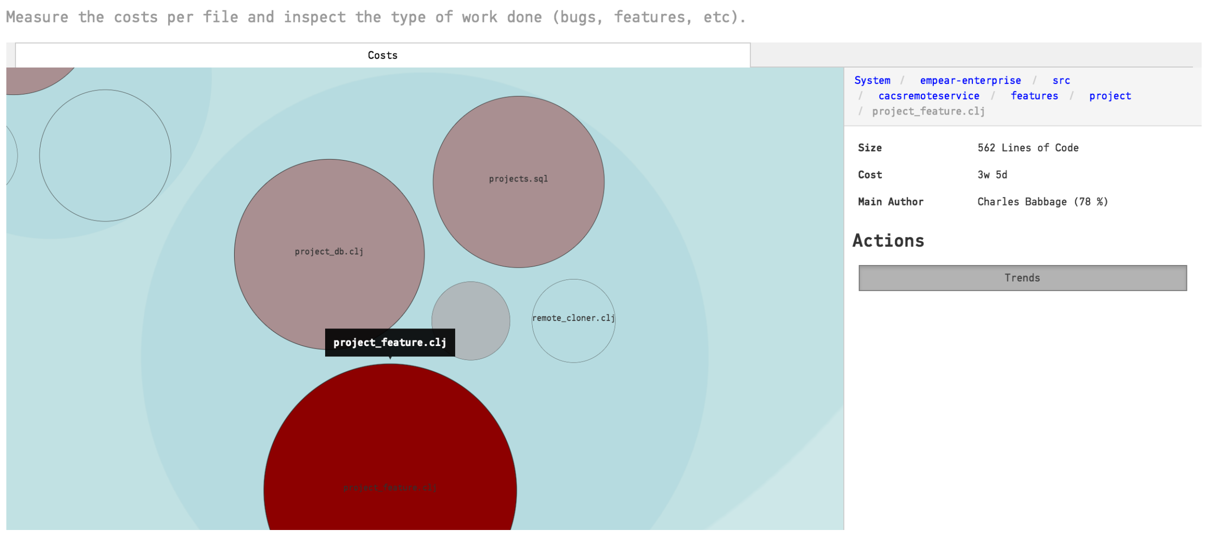
CodeScene bridges this chasm by introducing a suite of project management metrics. These metrics combines our existing version-control measures with data from Jira. This gives you Hotspots measured by cost rather than the more technical change frequency metric. It also gives you trends in both your costs 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.

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. 4.115


Measure the costs per file and inspect the type of work done (bugs, features, etc).

System / empear-enterprise / src / cacsremoteservice / features / project / project_feature.clj

Size 562 Lines of Code

Cost 3w 5d

Main Author Charles Babbage (78 %)

Fig. 4.115: A Hotspot analysis by cost lets you see where you spend most time.

As you see in Fig. 4.115, most time is spent in a module named *project_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 *Trends*.

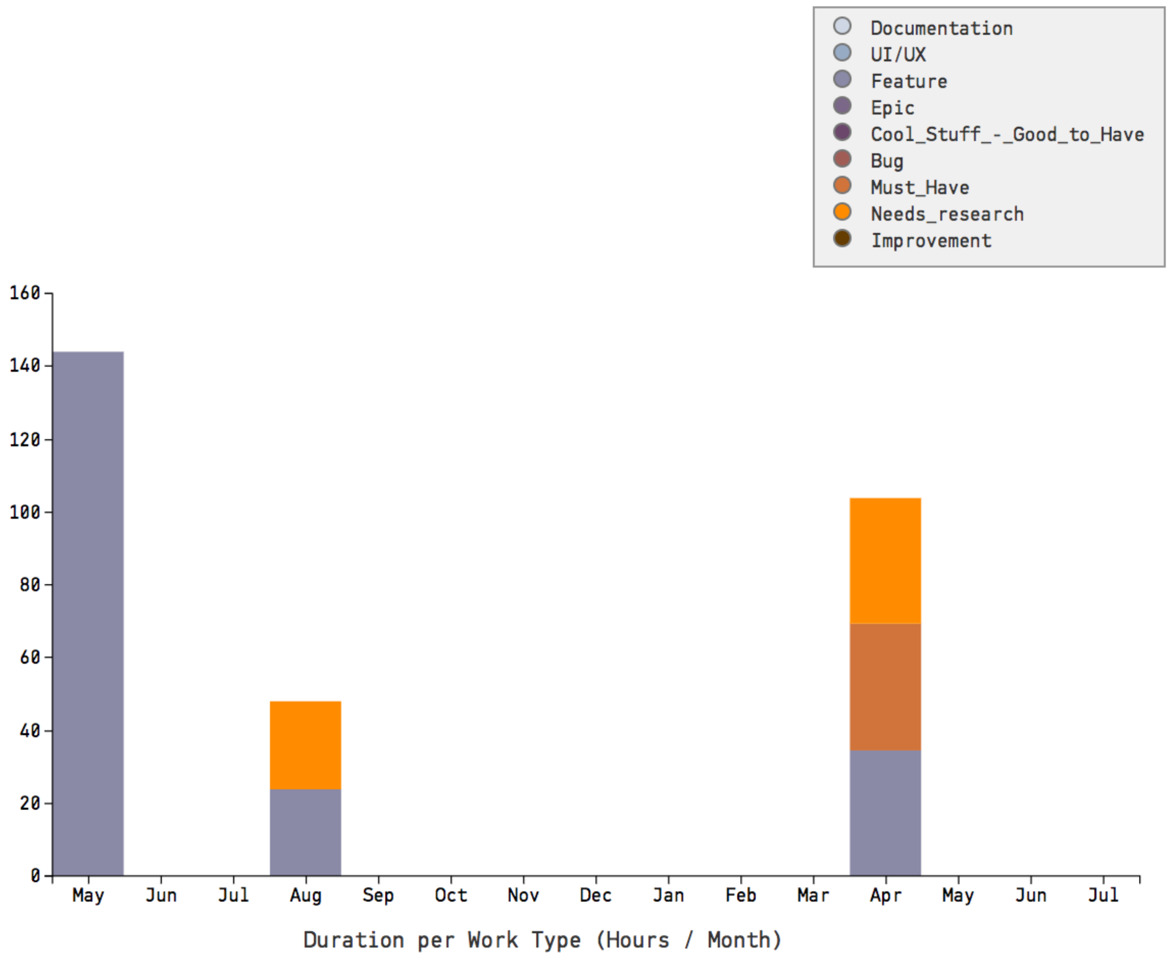


Fig. 4.116: Use the trends in type of work to see where your time is spent.

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, as illustrated in Fig. 4.116.
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. 4.117.

Measure Costs and Activity on Sub-Systems

In many systems the semantically interesting unit isn't individual files but rather sub-systems and components. Thus, CodeScene calculates the same cost metrics on an architectural level too. All you have to do is to enable your architectural analyses.

This kind of information gives you an overview of the costs on the sub-system level, and represents information that is relevant to non-technical managers too. Thus, use the analysis on this level to bridge the gap between the technical side of the organization and the business side by letting everyone share a common picture of how the system evolves.

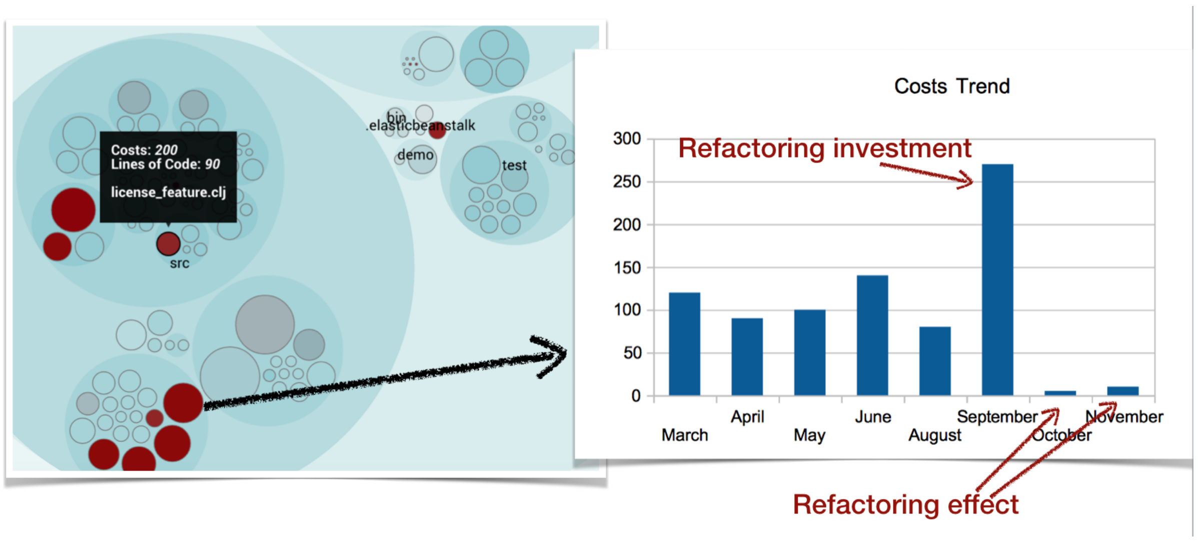
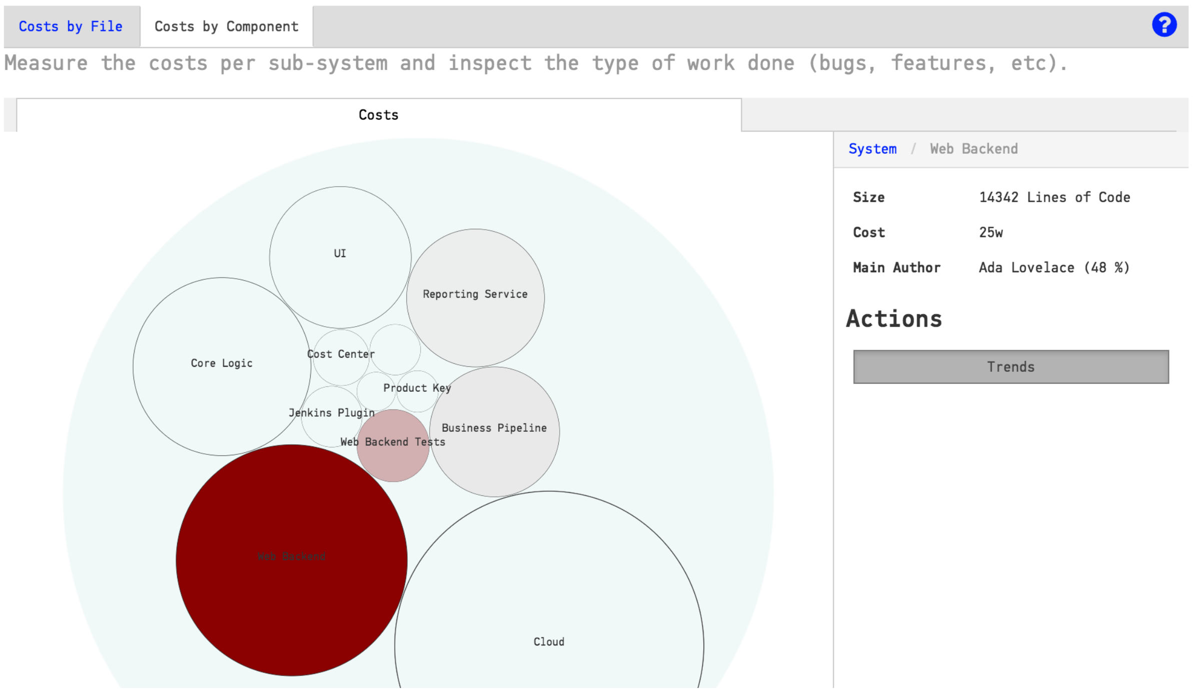


Fig. 4.117: Use the Cost Trends to measure the effect of improvements.



Measure the costs per sub-system and inspect the type of work done (bugs, features, etc).

Size	14342 Lines of Code
Cost	25w
Main Author	Ada Lovelace (48 %)

Fig. 4.118: Calculate hotspots by costs on architectural level

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. Note that the architectural level analyses mitigate this issue as they show the aggregated costs.

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.

4.5.2 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.

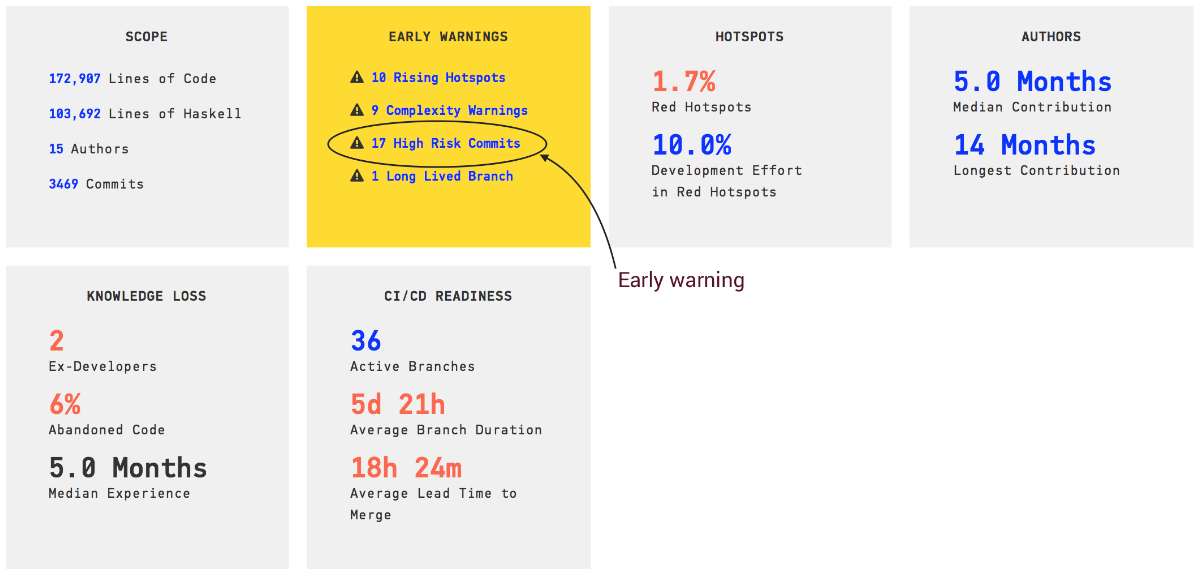
The risk classification that you'll see in CodeScene always combines these technical and social dimensions.

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. 4.119.



SCOPE

172,907 Lines of Code

103,692 Lines of Haskell

15 Authors

3469 Commits

EARLY WARNINGS

⚠ 10 Rising Hotspots

⚠ 9 Complexity Warnings

⚠ 17 High Risk Commits

⚠ 1 Long Lived Branch

HOTSPOTS

1.7%
Red Hotspots

10.0%
Development Effort in Red Hotspots

AUTHORS

5.0 Months
Median Contribution

14 Months
Longest Contribution

KNOWLEDGE LOSS

2
Ex-Developers

6%
Abandoned Code

5.0 Months
Median Experience

CI/CD READINESS

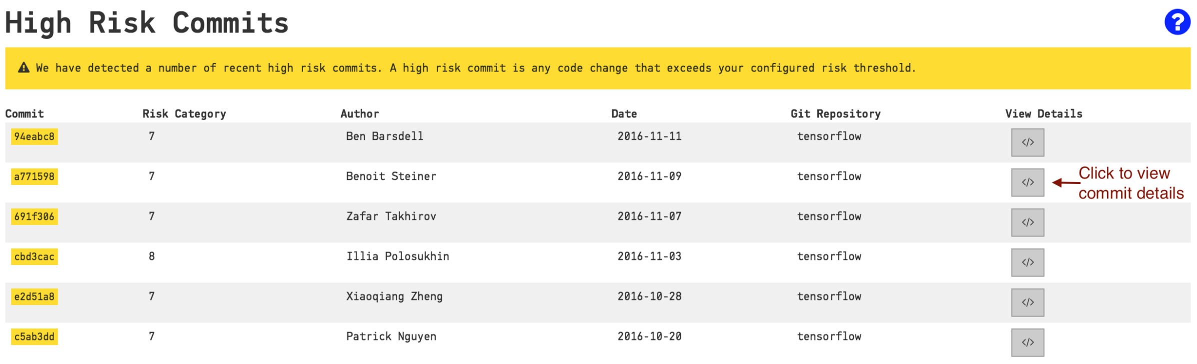
36
Active Branches

5d 21h
Average Branch Duration

18h 24m
Average Lead Time to Merge

Fig. 4.119: An Early Warning for recent high risk commits.

Click on the early warning shown in Fig. 4.119 to view the commit details as illustrated in Fig. 4.120.



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. 4.120: Inspect the details of each recent high risk commit.

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.

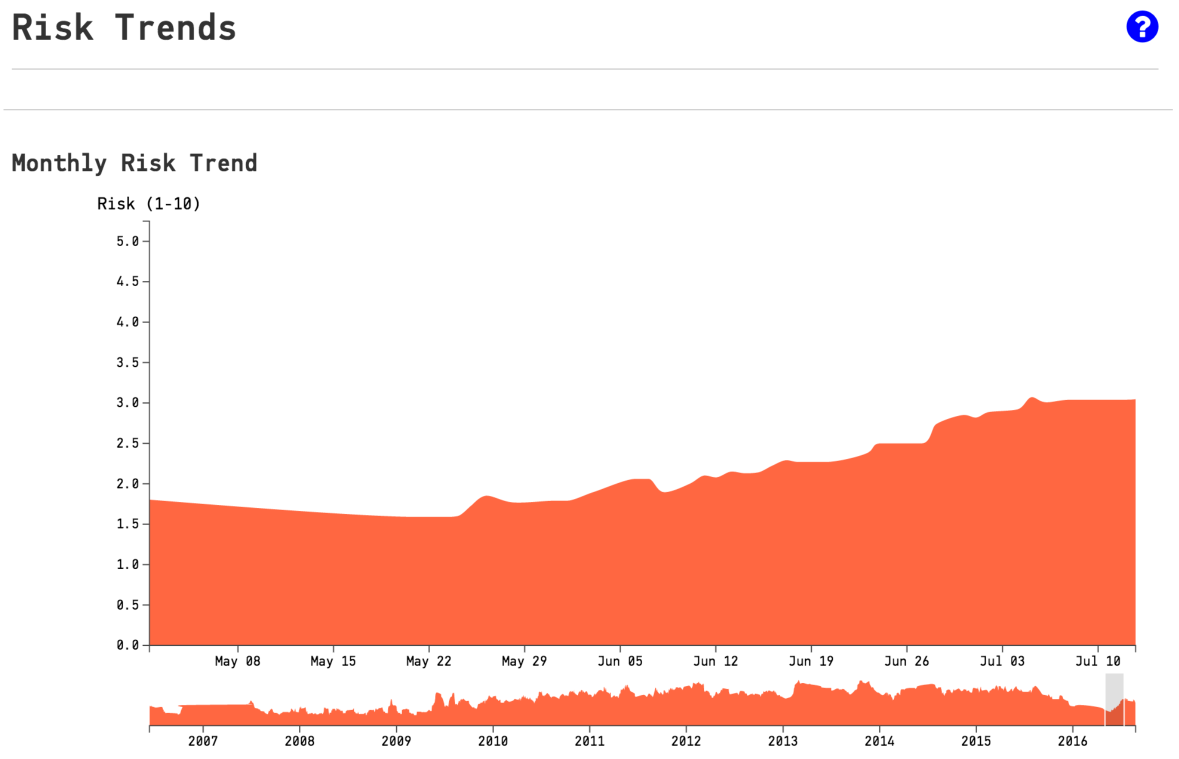


Fig. 4.121: The risk trend shows the average risk in the evolution of your codebase.

The example in Fig. 4.121 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.

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.

4.6 Continuous Integration and Code Review API

4.6.1 CI/CD Integration with CodeScene's Delta Analysis

CodeScene identifies and prioritizes technical debt, while at the same time uncovering and measuring social factors of the organization behind the system. The earlier you can react to any potential finding, the better. That's why CodeScene offers integration points that let you incorporate the analysis results into your build pipeline.

The purpose of a Delta Analysis is to:

- Prioritize code reviews based on the risk of the commits.

- Specify quality gates for the goals specified on identified hotspots using CodeScene's Intelligent Notes.
- Specify quality gates that trigger in case the Code Health of a hotspot declines.
- Get early warnings such a complexity trend increases and detect the absence of expected change coupling.

What are the Pre-Requisites for a Delta Analysis?

A Delta Analysis is always relative to a full analysis. CodeScene will use the latest completed analysis as a baseline for the Delta Analysis.

This is why we recommend that you configure your analysis to run at least once a day. On projects with more contributors and high commit frequencies you want to schedule CodeScene to run a full analysis each hour. We'd say that any project with more than 10 commits per day should run the analysis frequently.

Integrate CodeScene in your Continuous Integration Pipeline

CodeScene provides a REST API that lets you integrate the analysis results in a continuous integration pipeline and/or as robot comments in a code review tool like Gerrit.

CodeScene's REST API provides a special type of analysis called a *Delta Analysis*. A Delta Analysis is fast, it usually just takes a few seconds to run, and is used to get early feedback on a pull request or range of new commits.

Meet the Delta Analysis

A Delta Analysis is triggered by a pull request, a range of commits or a single commit; You decide through the API. Each time you trigger a Delta Analysis, CodeScene calculates the following information:

1. *Delivery risk of the suggested change set.* The risk classification is relative to the *risk profile* for your codebase as described in *Risk Analysis* (page 121). Use this information to prioritize code reviews and to decide upon delivery risks.

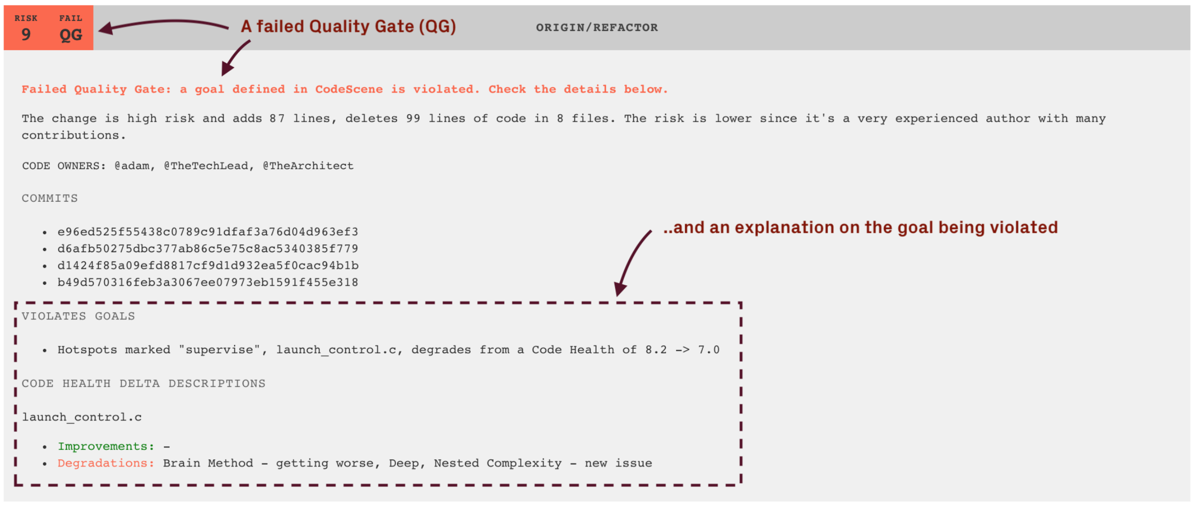
2. *Status of Quality Gates.* There are two independent – but related – quality gates. The first and most important one ensures that none of the goals you have specified via Intelligent Notes (see *Augmented Analysis with Intelligent Notes* (page 54)) are violated. The second quality gate lets you catch code that declines in Code Health directly in the CI/CD pipeline.

3. *Early detection of Complexity Trend Warnings.* CodeScene already provides an early warning in case the code complexity starts to rise in a Hotspot. Now the delta analysis can catch such complexity patterns based on the changes in a pull request. That provides a great opportunity to refactor the code before delivering it to your main branch.

4. *Suggests absent change patterns.* This analysis identifies change sets where an expected temporal coupling is absent. If a cluster of files have changed together for a long time they are intimately related. This warning fires when such a temporal change pattern is broken. Please note that this may be good - we've refactored something - but it may also be a sign of omission and a potential bug. As a consequence, this warning is based on a self-correcting algorithm; If you keep ignoring the warning it will go away automatically as the temporal coupling decreases below the thresholds.

The screenshot in Fig. 4.122 shows an example of a delta analysis result. This information is consumed and integrated via the REST API for delta analyses that we'll discuss soon.

Please note that future releases of CodeScene will expand the Delta Analysis capabilities. Our plan is to provide even more detailed information that helps you get the most out of your time.


RISK
9
FAIL
QG
A failed Quality Gate (QG)
ORIGIN/REFACTOR
..and an explanation on the goal being violated

Failed Quality Gate: a goal defined in CodeScene is violated. Check the details below.

The change is high risk and adds 87 lines, deletes 99 lines of code in 8 files. The risk is lower since it's a very experienced author with many contributions.

CODE OWNERS: @adam, @TheTechLead, @TheArchitect

COMMITS

- e96ed525f55438c0789c91dfaf3a76d04d963ef3
- d6afb50275dbc377ab86c5e75c8ac5340385f779
- d1424f85a09efd8817cf9d1d932ea5f0cac94b1b
- b49d570316feb3a3067ee07973eb1591f455e318

VIOLATES GOALS

- Hotspots marked "supervise", launch_control.c, degrades from a Code Health of 8.2 -> 7.0

CODE HEALTH DELTA DESCRIPTIONS

launch_control.c

- Improvements: -
- Degradations: Brain Method - getting worse, Deep, Nested Complexity - new issue

Fig. 4.122: A Delta Analysis gives you early warnings, actionable quality gates, and detects high risk changes (example from Jenkins).

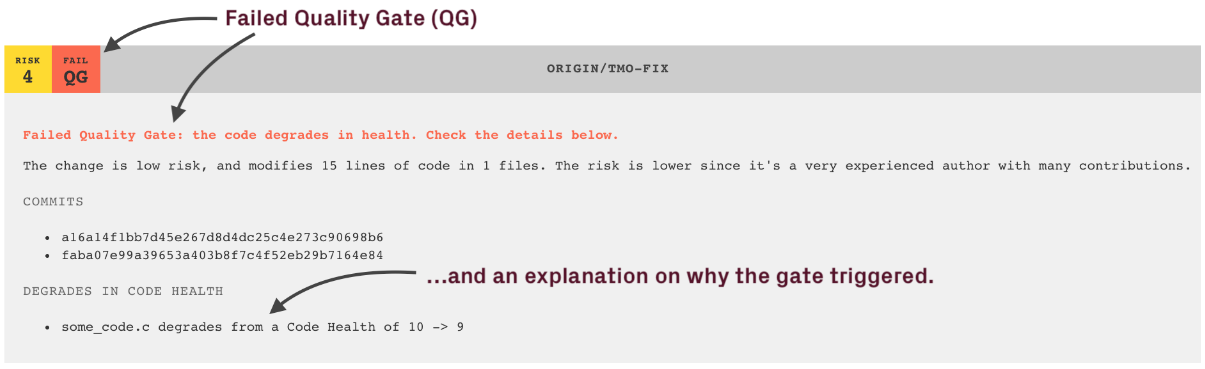
Use a Delta Analysis to Save Time in Code Reviews

The main advantage of a delta analysis if that it lets you react to potential problems early. But there's a potentially large saving at the other end of the spectrum too; Instead of treating all pull requests as equals, CodeScene's risk classification lets you prioritize your code reviews and focus your time where (and when) it's likely to be needed the most. Code reviewer fatigue is a real thing, so let's use our review efforts wisely.

Code review tools like Gerrit lets you select a label. For example, you specify a label that either allows or blocks the change. In addition you may select a label as an opinion (*+1* and *-1* in Gerrit).

When you integrate CodeScene with Gerrit, it's our recommendation that you map CodeScene's risk classification to an automated *+1* or *-1*. For example, all commits below the risk category *3* may be *+1*, which indicates to the reviewers how much time they need to spend on this review.

In addition, the delta analyses lets you auto-detect files that seem to degrade in quality through issues introduced in the current commit or pull request. This is done by calculating code biomarkers (see *Explore your Code's Biomarkers* (page 49)), which are then supervised for their trend as shown in Fig. 4.123.



Failed Quality Gate: the code degrades in health. Check the details below.

The change is low risk, and modifies 15 lines of code in 1 files. The risk is lower since it's a very experienced author with many contributions.

COMMITS

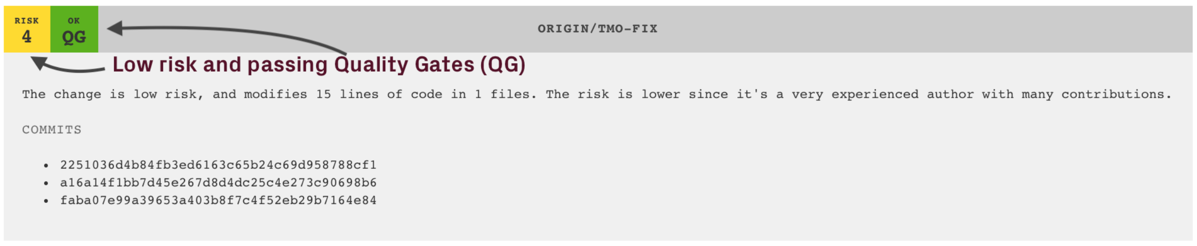
- a16a14f1bb7d45e267d8d4dc25c4e273c90698b6
- faba07e99a39653a403b8f7c4f52eb29b7164e84

DEGRADES IN CODE HEALTH

- some_code.c degrades from a Code Health of 10 -> 9

Fig. 4.123: A Delta Analysis detects degrading Code Health.

CodeScene's delta analysis works in the other direction too; It's not only about spotting problems. If you enable the quality gates and work pro-actively with Intelligent Notes in CodeScene, then you can allocate less time on the features with 1) low risk, and 2) passing gates, as shown in Fig. 4.124.



The change is low risk, and modifies 15 lines of code in 1 files. The risk is lower since it's a very experienced author with many contributions.

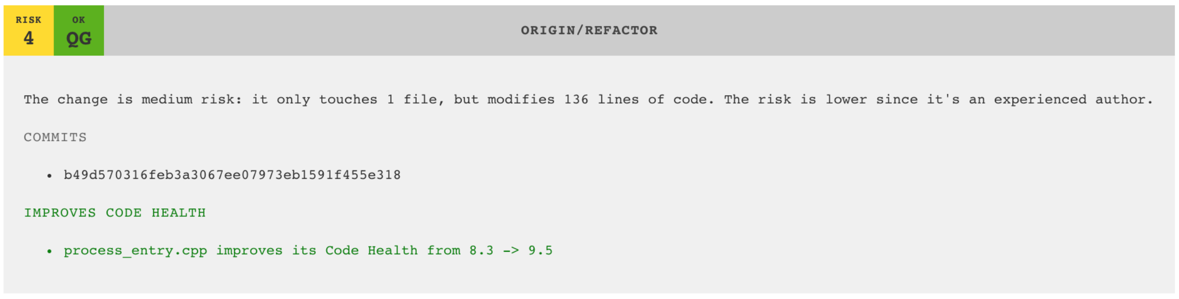
COMMITS

- 2251036d4b84fb3ed6163c65b24c69d958788cf1
- a16a14f1bb7d45e267d8d4dc25c4e273c90698b6
- faba07e99a39653a403b8f7c4f52eb29b7164e84

Fig. 4.124: Spend less time on reviewing code with low risk and passing quality gates.

Keep up the Good

CodeScene also detects hotspots that improve their code health. This information is included as a positive reinforcement intended to show the effect of the current changeset on the overall code health. Fig. 4.125 shows an example where a hotspot is successfully refactored – an occasion to celebrate!



The change is medium risk: it only touches 1 file, but modifies 136 lines of code. The risk is lower since it's an experienced author.

COMMITS

- b49d570316feb3a3067ee07973eb1591f455e318

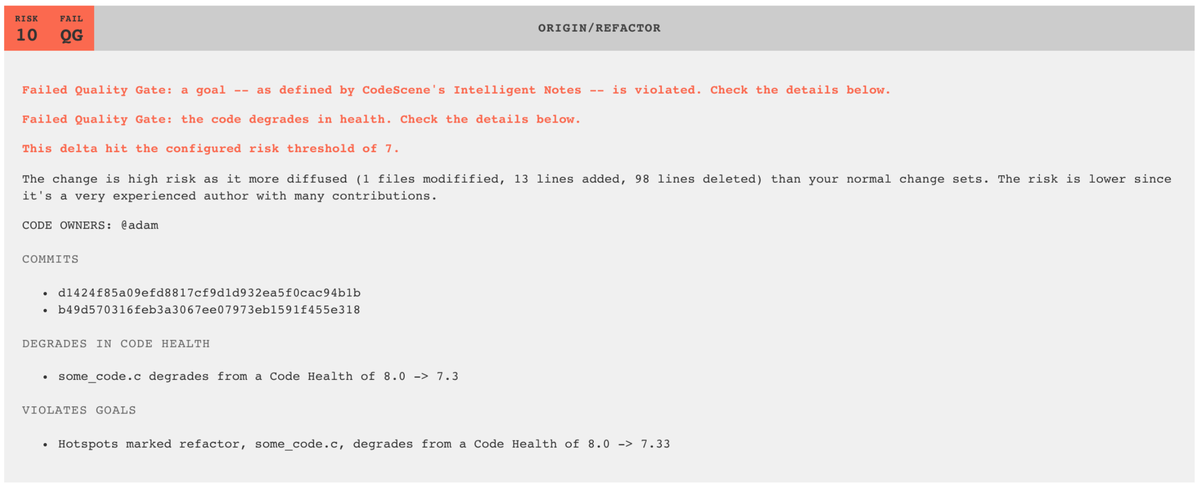
IMPROVES CODE HEALTH

- process_entry.cpp improves its Code Health from 8.3 -> 9.5

Fig. 4.125: Measure code improvements as part of the CI/CD pipeline to reinforce a positive trend.

Notify the Code Owners on Failed Quality Gates

As discussed in *Knowledge Distribution* (page 105), CodeScene parses and includes code ownership information when present. If you have this feature enabled, then CodeScene will include a mention of the code owners should a quality gate fail. You see an example in Fig. 4.126.



Failed Quality Gate: a goal -- as defined by CodeScene's Intelligent Notes -- is violated. Check the details below.

Failed Quality Gate: the code degrades in health. Check the details below.

This delta hit the configured risk threshold of 7.

The change is high risk as it more diffused (1 files modified, 13 lines added, 98 lines deleted) than your normal change sets. The risk is lower since it's a very experienced author with many contributions.

CODE OWNERS: @adam

COMMITS

- d1424f85a09efd8817cf9d1d932ea5f0cac94b1b
- b49d570316feb3a3067ee07973eb1591f455e318

DEGRADES IN CODE HEALTH

- some_code.c degrades from a Code Health of 8.0 -> 7.3

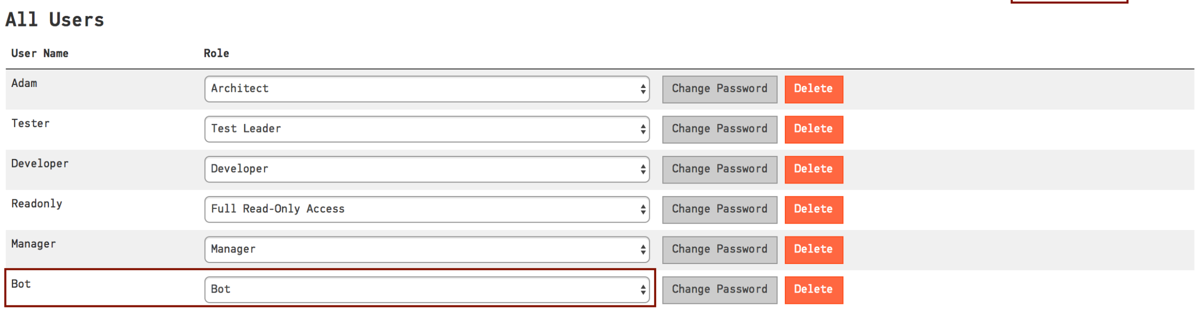
VIOLATES GOALS

- Hotspots marked refactor, some_code.c, degrades from a Code Health of 8.0 -> 7.33

Fig. 4.126: CodeScene notifies code owners when a quality gate fails.

The REST API for Delta Analyses

CodeScene lets you create a special *Bot* user role intended to consume the REST API. Login as administrator and create a Bot user for each of your integration points as illustrated in Fig. 4.127

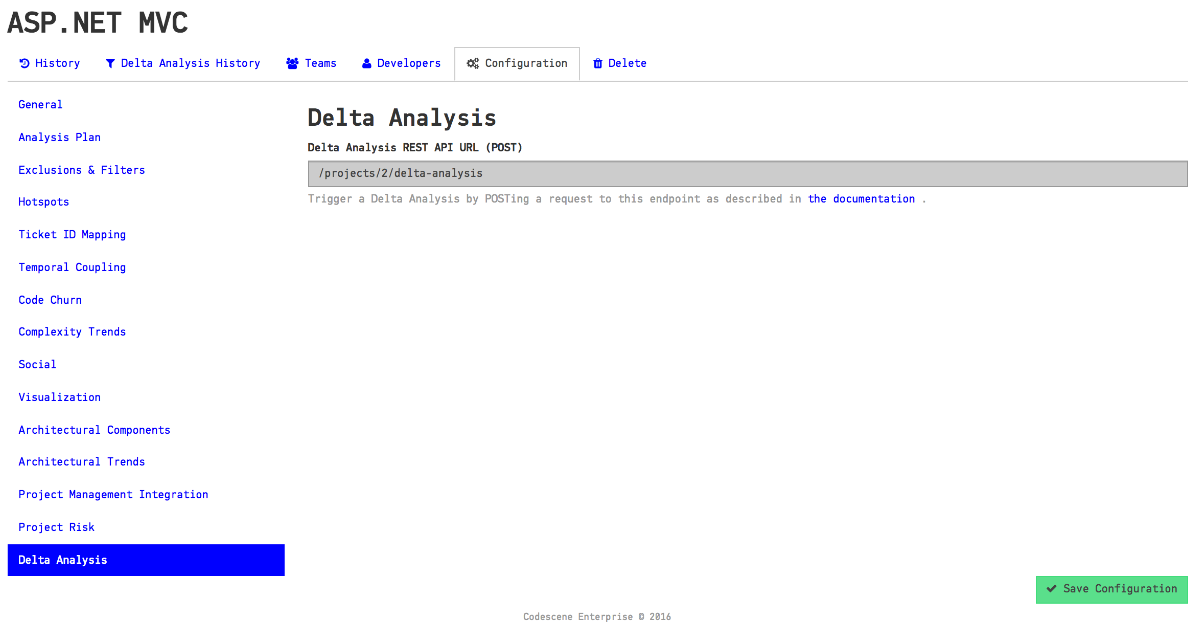


All Users

User Name	Role		
Adam	Architect	Change Password	Delete
Tester	Test Leader	Change Password	Delete
Developer	Developer	Change Password	Delete
Readonly	Full Read-Only Access	Change Password	Delete
Manager	Manager	Change Password	Delete
Bot	Bot	Change Password	Delete

Fig. 4.127: Configure a Bot user for each of your integration points.

You trigger a Delta Analysis by POSTing a request to the REST endpoint specified in your analysis configuration as illustrated in Fig. 4.128.



Delta Analysis

Delta Analysis REST API URL (POST)

/projects/2/delta-analysis

Trigger a Delta Analysis by POSTing a request to this endpoint as described in the documentation .

Fig. 4.128: Your analysis configuration specifies the REST endpoint to trigger a delta analysis.

The payload of the POST request specifies two required fields:

1. *commits*: This is a JSON array containing one or more commits. CodeScene will run a delta analysis on all these commits by considering them as a single unit of work.
2. *repository*: Specifies the Git repository where the *commits* that you want to analyse are located. You need to specify the repository name since an analysis project may contain multiple Git repositories.

Other optional parameters:

- *coupling_threshold_percent*: Specifies minimal temporal coupling for the "Absence of Expected Change" warning. Default is 80 (%).
- *use_biomarkers*: Instructs CodeScene to look for degrading code health, and enables reporting of the quality gates state. Note that this requires that the biomarkers are enabled for the analysis project.

Let's say that we have created an analysis project by specifying a Git remote:

https://github.com/PHPOffice/PhpSpreadsheet.git

In this case, the *repository* payload parameter is *PhpSpreadsheet* (strip the *.git* extension). If we want to simulate a delta analysis of the commit designated by the hash *99e5a8e919e1f7b83371a8a586fd6d7875f63583* we issue the following request:

```
curl -X POST -d '{"commits": ["149f9e6"], "repository": "PhpSpreadsheet"}' http://
↪localhost:3003/projects/64/delta-analysis -u 'CodeReview:MyPassword' -H "content-type:␣
↪application/json"
```

You can also specify a custom temporal coupling threshold:

```
curl -X POST -d '{"commits": ["149f9e6"], "repository": "PhpSpreadsheet", "coupling_threshold_
↪percent": 50}' http://localhost:3003/projects/64/delta-analysis -u 'CodeReview:MyPassword' -
↪H "content-type: application/json"
```

Finally, you can enable biomarkers to detect potential code quality problems early:

```
curl -X POST -d '{"commits": ["149f9e6"], "repository": "PhpSpreadsheet", "use_biomarkers":␣
↪true}' http://localhost:3003/projects/64/delta-analysis -u 'CodeReview:MyPassword' -H
↪"content-type: application/json"
```

The example assumes that 1) CodeScene runs on localhost and 2) we have configured a user named *CodeReview*.

Notes to Windows Users: The curl syntax above won't work on Windows unless you escape the payload properly. We recommend that you use Fiddler instead of curl if you want to test the API on Windows.

Once you've issued the POST request above, CodeScene's Delta Analysis will respond with the following JSON document:

```
{"version":"1",
 "url":"/projects/64/delta-analysis/75686456d695d60d99a7cd73302f83606c8a8efc",
 "view":"/64/delta-analysis/view/75686456d695d60d99a7cd73302f83606c8a8efc",
 "result":{"risk":3,
          "description":"The change is low risk as it touches less code than your typical␣
↪change set",
          "improvements":["Connector.java improves its code health from 5.6 -> 6.2"],
          "quality-gates":{"degrades-in-code-health":false,
                           "violates-goal":false},
          "warnings":[]}}
```

The parameters in the response carry the following meaning:

- *version*: This is the version of the REST API and will change in future versions of CodeScene.
- *url*: This URL points to the Delta Analysis resource in CodeScene. You can fetch it with an HTTP GET request at any time and it will return the same result document.
- *view*: Points to the page in CodeScene that contains the graphical representation of the result as illustrated in Fig. 4.122.
- result*: This JSON object contains four fields:*

  - *risk* is the risk classification of the commit(s), range 1-10.
  - *description* is a human readable interpretation of the risk calculated by CodeScene's machine learning algorithms.
  - *improvements* is a positive reinforcement that shows code that improves its health.
  - *quality-gates* specifies the state of the two gates that can, optionally, be checked and enforced in the requesting build pipeline.

  – *warnings* specify any early warnings like Complexity Trend increases. In this case it's a low risk commit without any early warning.

Now, let's look at a more complex result. In this case a new developer has made a modification to one of the top Hotspots on a separate branch. The Delta Analysis reports the following results:

```
{"version":"1",
 "url":"/projects/2/delta-analysis/43cc8a146cc0957f2fcb4b09ae3dee71d5a5cf2e",
 "view":"/2/delta-analysis/view/43cc8a146cc0957f2fcb4b09ae3dee71d5a5cf2e",
 "result":{"risk":10,
           "description":"The change is high risk as it is a more wide-spread
                          change (1232 lines of code in 32 files) than your
                          typical change patterns.
                          The risk increases as the author has somewhat lower
                          experience in this repository.",
           "improvements":[],
           "quality-gates":{"degrades-in-code-health":false,
                            "violates-goal":false},
           "warnings":[
             {"category":"Modifies Hotspot",
              "details":["mvc/src/Microsoft.AspNetCore.Mvc.TagHelpers/LinkTagHelper.cs"]},
             {"category":"Absence of Expected Change Pattern",
              "details":["mvc/src/Microsoft.AspNetCore.Mvc.TagHelpers/ScriptTagHelper.cs"]}]}}
```

We see that CodeScene reports a high risk of *10*. We also note that CodeScene calls our attention to the modified Hotspot. We use this information to review the change more carefully. Finally we note that CodeScene detects the absence of an expected change pattern. In this codebase, the *LinkTagHelper.cs* is usually changed together with the *ScriptTagHelper.cs* file. Since that wasn't the case here, CodeScene informs us about the omission so that we can investigate it and catch a potential bug early.

Finally, let's see how a failed code quality gate looks:

```
{"version":"1",
 "url":"/projects/2/delta-analysis/43cc8a146cc0957f2fcb4b09ae3dee71d5a5cf2e",
 "view":"/2/delta-analysis/view/43cc8a146cc0957f2fcb4b09ae3dee71d5a5cf2e",
 "result":{"risk":10,
           "description":"The change is high risk as it is a more wide-spread
                          change (1232 lines of code in 32 files) than your
                          typical change patterns.
                          The risk increases as the author has somewhat lower
                          experience in this repository.",
           "improvements":[],
           "quality-gates":{"degrades-in-code-health":true,
                            "violates-goal":true},
           "code-owners-for-quality-gates":["@TheTester" "@TheMicroManager"],
           "warnings":[
             {"category":"Degrades in Code Health",
              "details":["ViewComponentResultTest.cs degrades from a Code Health of 10.0 -> 9.0
↪"
                         "ControllerActionInvokerTest.cs degrades from a Code Health of 5.0 ->␣
↪4.7"]},
             {"category":"Violates Goals",
              "details": ["Hotspots marked supervise, ControllerActionInvokerTest.cs, degrades␣
↪from a Code Health of 5.0 -> 4.67"]}]}}
```

The failed quality gates are indicated in the *quality-gates* field. We also note that CodeScene includes the *code-owners-for-quality-gates* field, which specifies the two owners – as read from a *CODEOWNERS* file – that are responsible for the code that failed the quality gates.

Delta Analysis in Offline Mode

Delta analysis is triggered via an API call and thus requires authentication. Since CodeScene checks the license on a remote license server with every authentication request, the delta analysis API call will fail if CodeScene can't reach the license server. If you don't have an Internet connection or you don't want to let CodeScene access the Internet, you need to specify *offline-mode* parameter:

```
curl -X POST -d '{"commits": ["149f9e6"], "repository": "PhpSpreadsheet"}' http://
↪localhost:3003/projects/64/delta-analysis?offline-mode=offline -u 'CodeReview:MyPassword' -H
↪"content-type: application/json"
```

Please note that there is a new *offline-mode=offline* parameter in the query string.

If you always run CodeScene in offline mode, you can also turn on *Global Offline Mode* in the configuration. With global offline mode, you don't have to append *offline-mode=offline* parameter to your delta analysis API URLs.

Read more about the limitations and usage in *Offline Mode* (page 6) documentation.

4.6.2 Branch Analyses

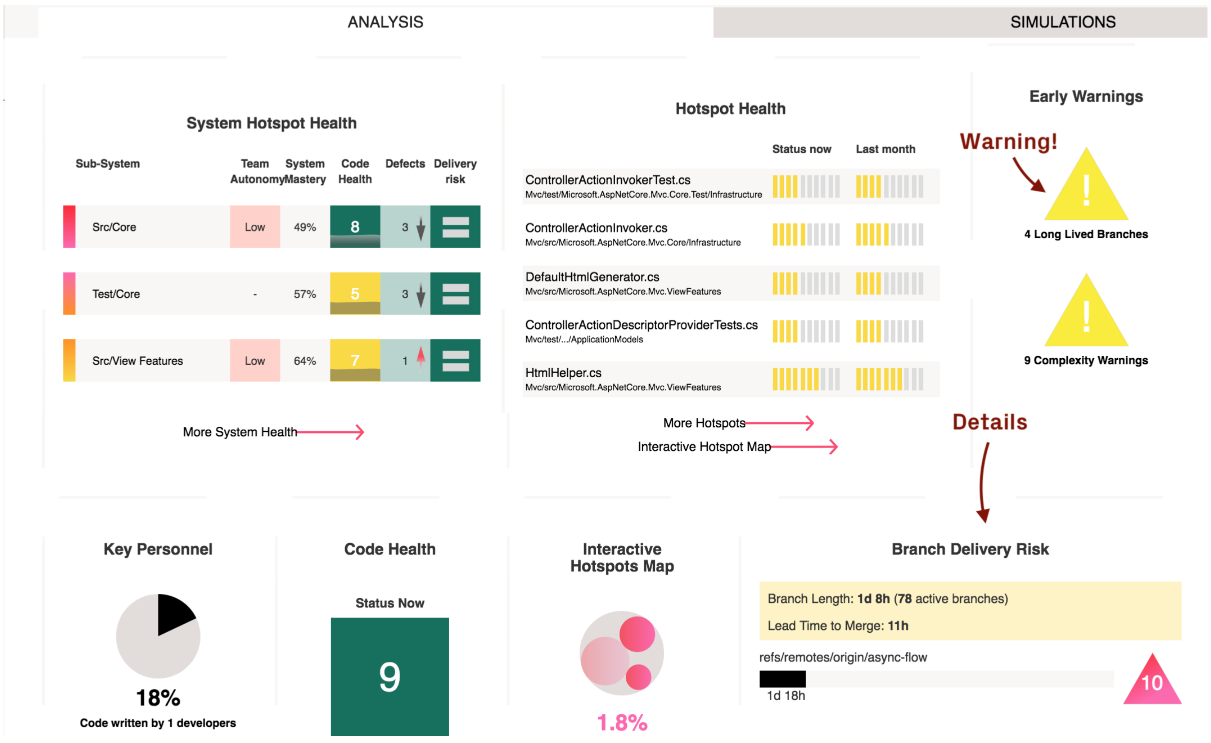
Many organizations are transitioning to short-lived feature branches and employ practices like continuous integration/delivery. To work in practice, branches have to be kept short-lived.

CodeScene introduces a new suite of analyses that measure branching activity, different lead times, and risks. This is information you can use to get insights into your CI/CD process, or to reason about delivery and development risks.

Note: You need to have at least version 2.15 of Git in order to enable the branch analyses.

Meet the Branch Measures

CodeScene presents a summary of the branch statistics on its dashboard as shown in Fig. 4.129.



Sub-System	Team Autonomy	System Mastery	Code Health	Defects	Delivery risk
Src/Core	Low	49%	8	3	
Test/Core	-	57%	5	3	
Src/View Features	Low	64%	7	1	

	Status now	Last month
ControllerActionInvokerTest.cs Mvc/test/Microsoft.AspNetCore.Mvc.Core.Test/Infrastructure		
ControllerActionInvoker.cs Mvc/src/Microsoft.AspNetCore.Mvc.Core/Infrastructure		
DefaultHtmlGenerator.cs Mvc/src/Microsoft.AspNetCore.Mvc.ViewFeatures		
ControllerActionDescriptorProviderTests.cs Mvc/test/.../ApplicationModels		
HtmlHelper.cs Mvc/src/Microsoft.AspNetCore.Mvc.ViewFeatures		

Branch Length: **1d 8h** (**78** active branches)

Lead Time to Merge: **11h**

refs/remotes/origin/async-flow

1d 18h

Fig. 4.129: An overview of the branch measures.

Branch Duration
: The calendar time from the first commit of a branch to the latest commit of the branch. For example, if the first commit on the feature branch *task-52-support-cherenkov-drive-mode* is made at 7am on Dec 1st, and the second commit is done at 10am on the same day, the branch duration at that point is three hours. High values indicate long-lived branches, which can indicate high risk.

Lead Time to Merge
: The calendar time between when the last commit of the branch is made, and when the branch is merged. It can be caused by waiting for code reviews or other tasks required before merging.

Contributing Authors
: The number of unique authors that have contributed to a branch. A high number could indicate a complicated feature, or that it contains many features from different authors.

By default, CodeScene calculates statistics for all branches that have been worked on during the past two months.

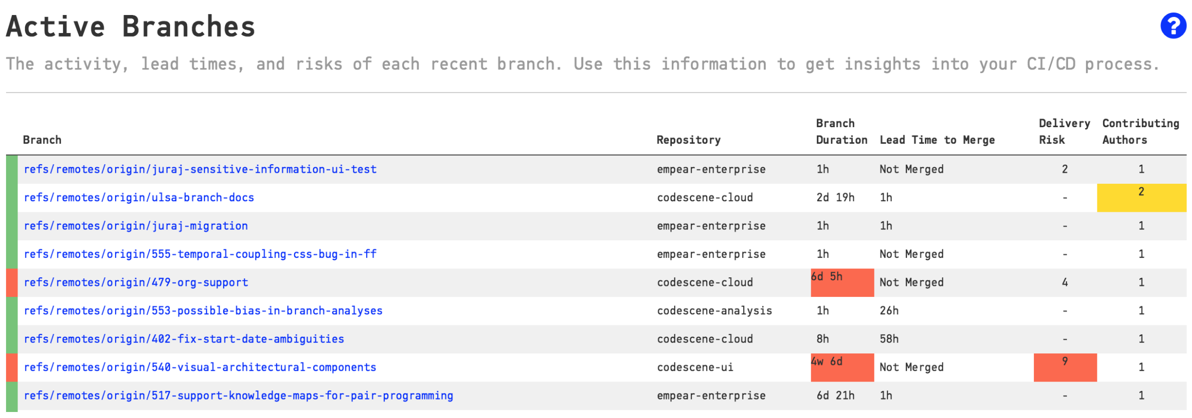
> *Note*: you can configure the time span for active branches as well as filter out specific branch names (e.g. long-lived release branches) in the project configuration.

Use this high-level overview to ensure you have a short *Branch Duration* and a short *Lead Time to Merge* for each branch. When a branch lives for too long, it puts your delivery at risk of merge conflicts and unexpected feature interactions.

As you see in the preceding figure, CodeScene also auto-detects early warnings for long lived branches. Use this information to either:

- Re-plan the scope: Sometimes it's just too much work in a single feature. Identifying a smaller feature set that you can deliver faster is one way to shorten the lead times and minimize risk.
- Prioritize verification activities: Use the early warning to focus extra code reviews and tests on the highlighted branches.

If you click on the branch tile on the dashboard, CodeScene displays a detailed view of each branch as shown in Fig. 4.130.



Active Branches

The activity, lead times, and risks of each recent branch. Use this information to get insights into your CI/CD process.

Branch	Repository	Branch Duration	Lead Time to Merge	Delivery Risk	Contributing Authors
refs/remotes/origin/juraj-sensitive-information-ui-test	empear-enterprise	1h	Not Merged	2	1
refs/remotes/origin/ulsa-branch-docs	codescene-cloud	2d 19h	1h	-	2
refs/remotes/origin/juraj-migration	empear-enterprise	1h	1h	-	1
refs/remotes/origin/555-temporal-coupling-css-bug-in-ff	empear-enterprise	1h	Not Merged	-	1
refs/remotes/origin/479-org-support	codescene-cloud	6d 5h	Not Merged	4	1
refs/remotes/origin/553-possible-bias-in-branch-analyses	codescene-analysis	1h	26h	-	1
refs/remotes/origin/402-fix-start-date-ambiguities	codescene-cloud	8h	58h	-	1
refs/remotes/origin/540-visual-architectural-components	codescene-ui	4w 6d	Not Merged	9	1
refs/remotes/origin/517-support-knowledge-maps-for-pair-programming	empear-enterprise	6d 21h	1h	-	1

Fig. 4.130: A detailed analysis of the work on each branch.

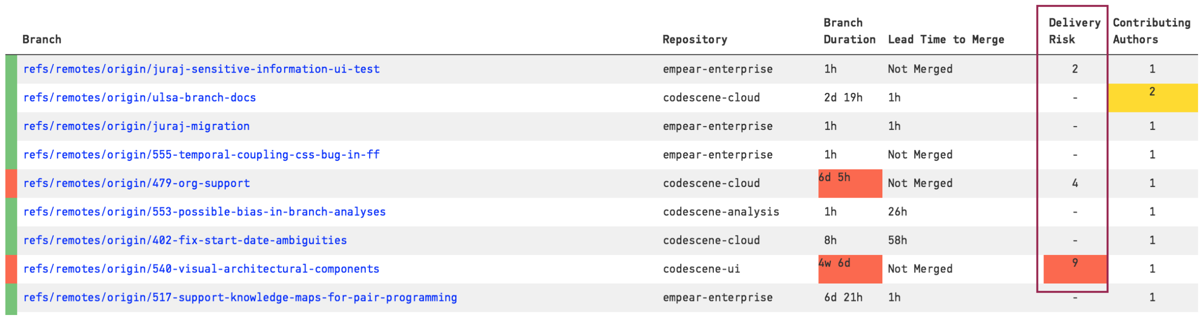
This information lets you identify early signs of trouble, such as long-lived branches, or branches that become congested by attracting contributions from several different authors.

Note that the thresholds used to trigger the early warnings are automatically deduced from your normal branching strategy; CodeScene warns when a branch deviates from your normal ways of working. As such, the warnings are relative to the patterns in your codebase.

You can see another example of such deviations from normal ways of working in Fig. 4.130: One of the branches has contributions from 5 different developers, which might put you at risk for parallel development. It's also a sign that there's too much development activity on that branch, so you could use this information to investigate the scope.

Detect Delivery Risks

CodeScene predicts the delivery risk of each active branch (i.e. unmerged work) as shown in Fig. 4.131.



Branch	Repository	Branch Duration	Lead Time to Merge	Delivery Risk	Contributing Authors
refs/remotes/origin/juraj-sensitive-information-ui-test	empear-enterprise	1h	Not Merged	2	1
refs/remotes/origin/ulsa-branch-docs	codescene-cloud	2d 19h	1h	-	2
refs/remotes/origin/juraj-migration	empear-enterprise	1h	1h	-	1
refs/remotes/origin/555-temporal-coupling-css-bug-in-ff	empear-enterprise	1h	Not Merged	-	1
refs/remotes/origin/479-org-support	codescene-cloud	6d 5h	Not Merged	4	1
refs/remotes/origin/553-possible-bias-in-branch-analyses	codescene-analysis	1h	26h	-	1
refs/remotes/origin/402-fix-start-date-ambiguities	codescene-cloud	8h	58h	-	1
refs/remotes/origin/540-visual-architectural-components	codescene-ui	4w 6d	Not Merged	9	1
refs/remotes/origin/517-support-knowledge-maps-for-pair-programming	empear-enterprise	6d 21h	1h	-	1

Fig. 4.131: Predict the delivery risk of each branch.

This *risk classification* predicts the risk for defects, and is given on the scale 1-10 where 10 is the highest risk.

Use this information to plan preventive measures such as extra code reviews and tests. You can also setup a separate CodeScene analysis and just focus on the work being done on the branch. In extreme cases you may chose to postpone the merge of such high-risk branches if you're close to a critical deadline.

Repo-based Projects

Branch Analyses are not currently supported in Repo-based projects. This is because Repo does not automatically check out a reference branch. For more on using Repo in CodeScene, see *Working with Repo* (page 156).

4.7 Delivery Performance

4.7.1 Measure Delivery Performance

Software development productivity is notoriously hard to measure. At the same time, it's important to be able to visualize the business impact of any technical debt that CodeScene detects, and it's also valuable to illustrate the effects of larger refactorings.

For that purpose, CodeScene includes a Delivery Performance module.

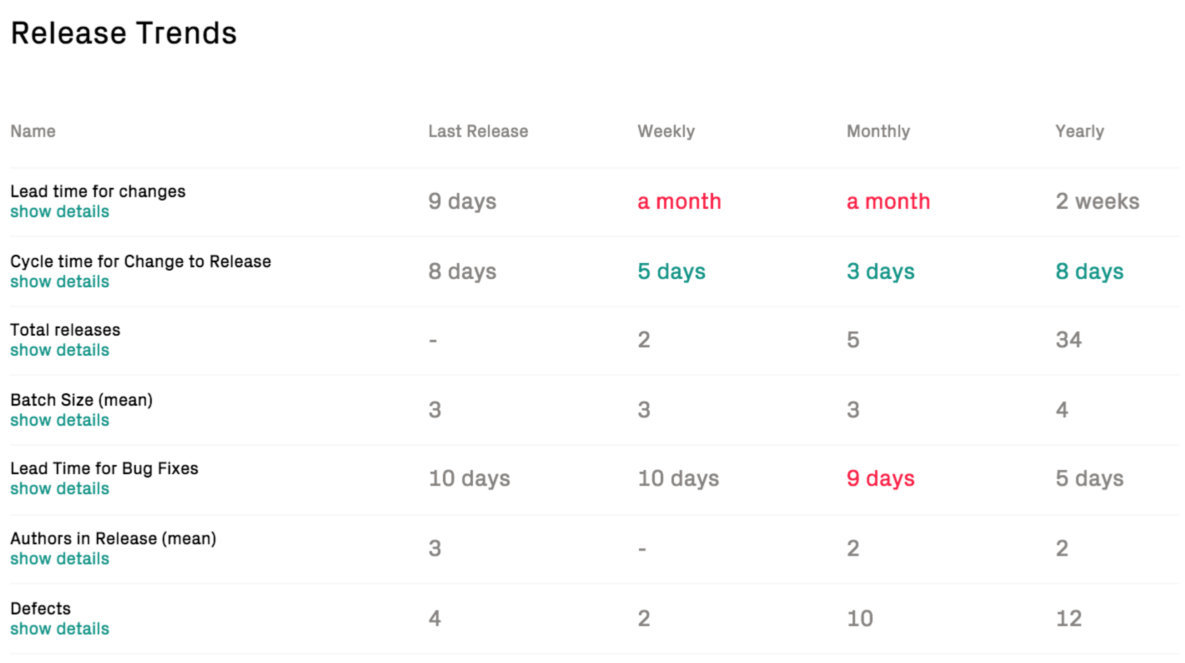
Delivery Performance Measures Lead Times and Defect Ratios

CodeScene's Delivery Performance module focuses on throughput and lead time metrics that are known to correlate with business values like time to market, customer satisfaction, and profitability (see Accelerate: The Science of Lean Software and DevOps by Forsgren, Humble, and Kim for a deeper read on this topic).

Like most CodeScene metrics, the emphasize is on trends over absolute values. That is, it's more important that you – as an organization – move in the desired direction than to compare your performance against any other entity in a different context. Let's look at an example in Fig. 4.132.

The metrics are measured as follows:

- Lead Time for Changes: The lead time starts when a Jira issue enters an in progress/development state, and ends when the feature is included in a release. This is typically the most important metrics since lead times correlate with time to market.
- Defects: It would be trivial to shorten lead times if we compromised quality. Hence, we measure the number of defects fixed in each release. An increase might signal positive aspects like improved testing, or – more frequently – a degradation in external quality.



Release Trends

Name	Last Release	Weekly	Monthly	Yearly
Lead time for changes show details	9 days	a month	a month	2 weeks
Cycle time for Change to Release show details	8 days	5 days	3 days	8 days
Total releases show details	-	2	5	34
Batch Size (mean) show details	3	3	3	4
Lead Time for Bug Fixes show details	10 days	10 days	9 days	5 days
Authors in Release (mean) show details	3	-	2	2
Defects show details	4	2	10	12

Fig. 4.132: CodeScene calculates trends based on the periodic releases.

- Cycle Time from Change to Release: This cycle time starts when a commit is made until the commit is included in a release. A long cycle time from change to release indicates a bottleneck in the delivery process such as in-efficent code reviews, over-reliance on manual verification, or simply to in-frequent releases.
- Lead Time for Bug fixes: Measures the time from opening a bug/defect in Jira until the bug fix is included in a release. A long lead-time for bug fixes might mean that users have to wait for a response to issues, which might lead to less customer satisfaction.
- Batch Size: Measures the number of Jira issues included in a release. Efficient delivery is much about minimizing the amount of parallel work and releasing new features in small batches; the larger our releases, the more expensive it is to debug and isolate potential failures.

Measure the Costs of Planned vs Unplanned Work

CodeScene's Jira integration (see *Integrate Jira Information into CodeScene* (page 29)) provides a detailed breakdown of the development costs. To complement that data, the Delivery Performance module provides an overview of those development costs categorized as *Planned* and *Unplanned* work:

The categorization is provided in the configuration. Typically, everything related to features or planned re-work is categorized as *Planned*, whereas bugs and service interruptions are *Unplanned*.

> The amount of Unplanned work indicates the unrealized potential that can be optimized in an organization's delivery efficiency; minimizing the amount of Unplanned work leads to more predictable progress and delivery pace.

Detailed Delivery Performance Graphs

All lead times and quality trends are complemented with interactive graphs that let you detect outliers and inspect specific releases.

The graphs are complemented with a trend showing the size of the development team over time. The rationale for including the scale of the team is that declining delivery performance trends often indicate on-boarding costs and/or increased communication and coordination needs (see *Delivery Effectiveness by Organizational Trends* (page 101) for suggested follow-up analyses).

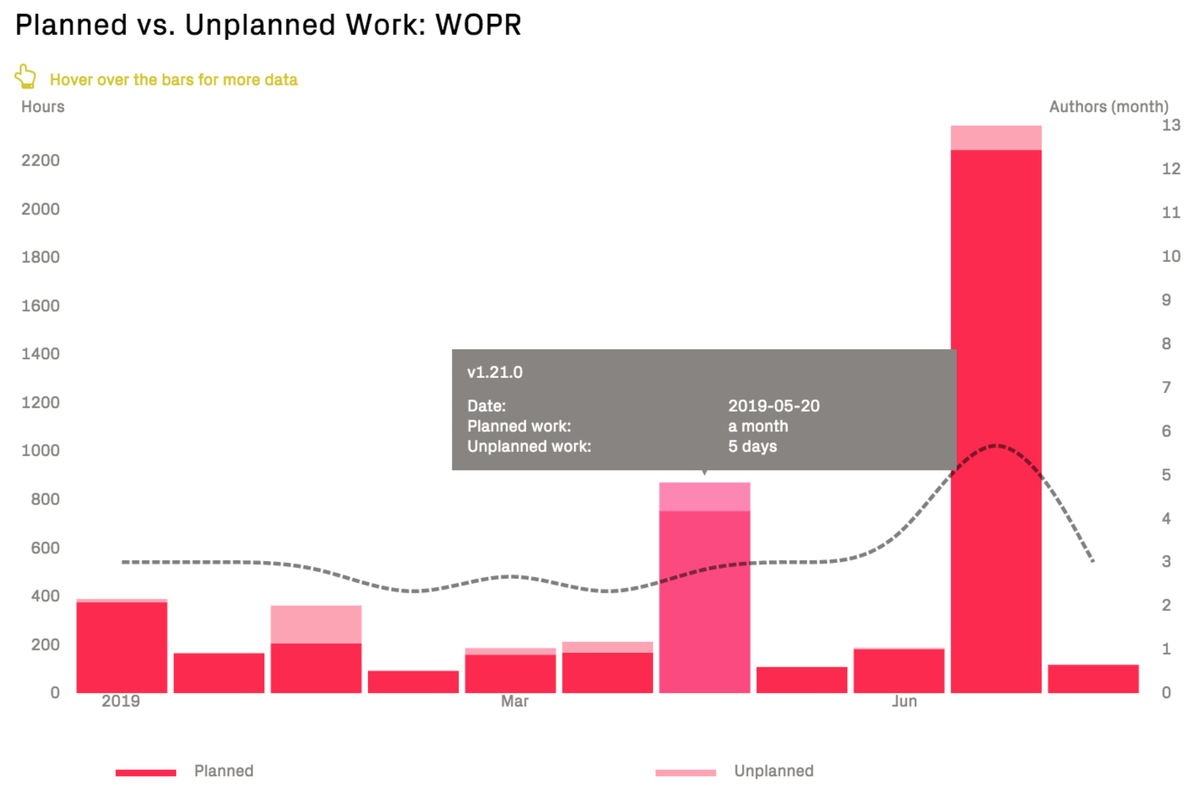


Fig. 4.133: Inspect how the development capacity is spent; do we spend our time mainly on Planned or Unplanned work?

Interpret the Graphs

Here's an example on a dramatic increase in lead times that indicate a delivery issue:

The previous graph should be an immediate call to arms to uncover the potential bottlenecks in the product development. For that purpose, turn to CodeScene's other metrics for organizations and technical debt. It's also worth pointing out that the batch size has been increasing significantly in the preceding graph; the popups show that the last release is almost 3 times as large as the previous release.

We can get more information on the batch size trends in the next graph:

Contrast the preceding trend with how the batch size looks in an organization that practices continuous delivery:

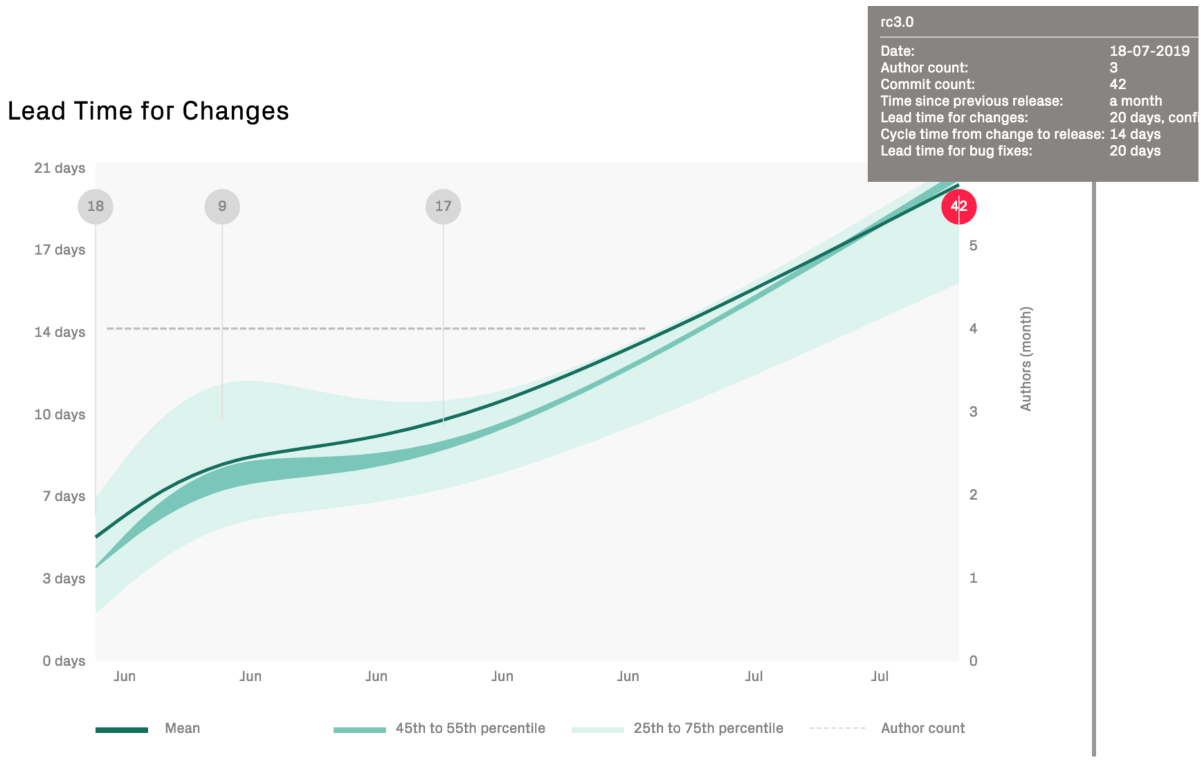
Small batch sizes has a large impact on delivery performance; the smaller releases we do, the easier it is to track down and isolate any errors or failures.

Use the Predictive Analytics as Early Warnings

CodeScene's delivery performance module also includes a set of predictive analytics that provide feedback on the work in progress. That is, CodeScene calculates metrics on the yet to be released commits and issues as shown in Fig. 4.137.

These predictions are based on patterns in how you – as an organization – has worked so far; that's how CodeScene knows that according to your typical ways of working, you should have released 9 days ago. Use such feedback as an indication that it might be time to cut the scope and wrap up the ongoing release.

The predictions on ongoing work serve as visual reminders on the importance of maintaining short lead times and to limit work in progress.



rc3.0

Date: 18-07-2019
Author count: 3
Commit count: 42
Time since previous release: a month
Lead time for changes: 20 days, confi
Cycle time from change to release: 14 days
Lead time for bug fixes: 20 days

Fig. 4.134: Increasing lead times might indicate either organizational issues, planning problems, or growing technical debt.

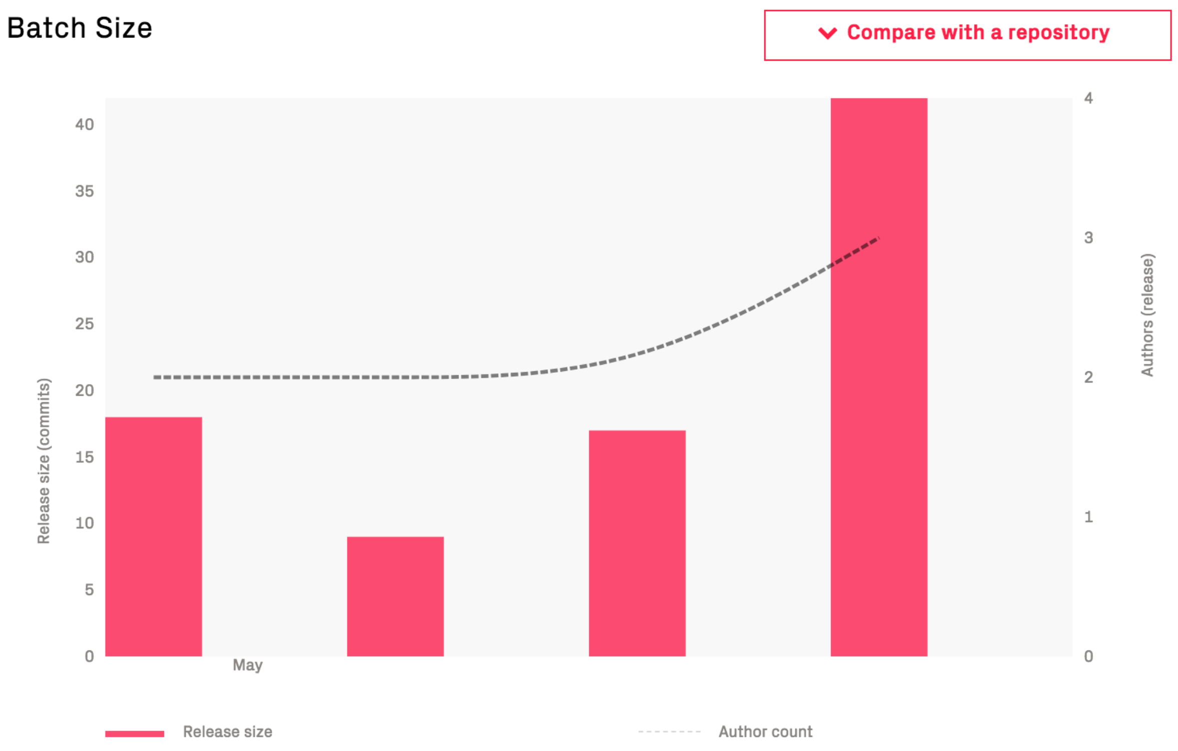


Fig. 4.135: Increasing batch sizes are an indication that too much work is included in each release.

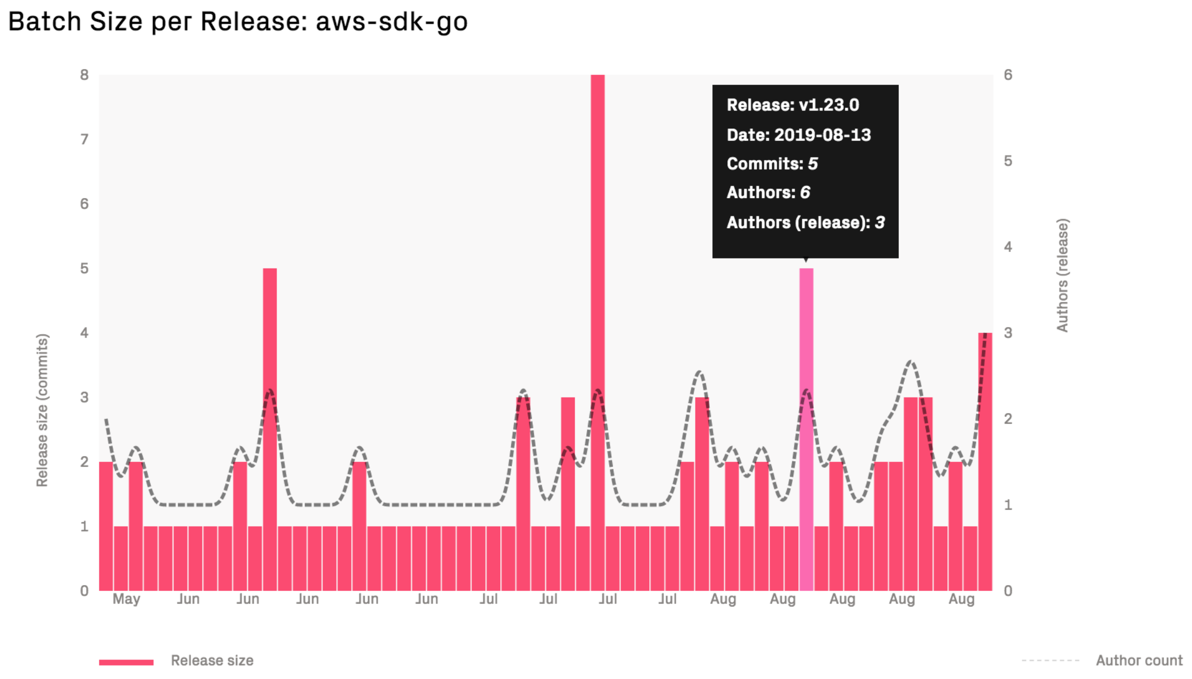
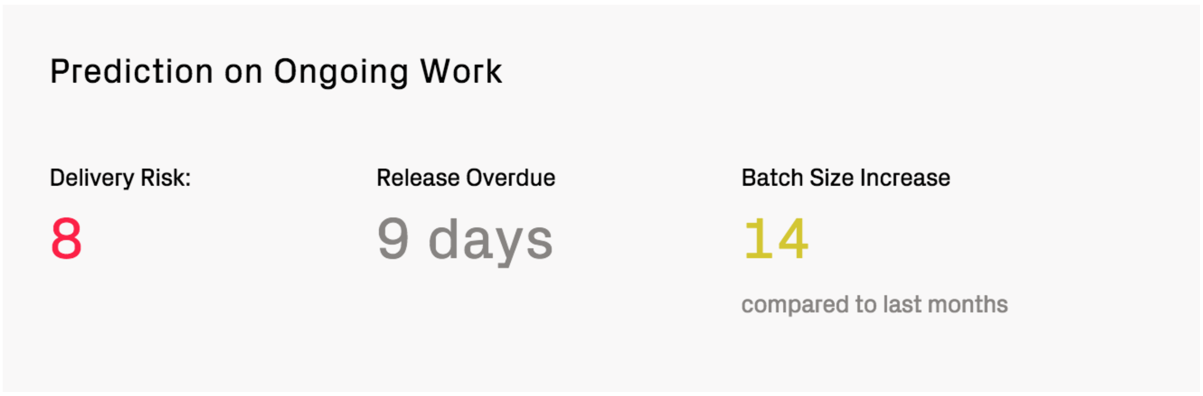


Fig. 4.136: *In Continuous Delivery, each feature is released when it's done.*



Prediction on Ongoing Work

Delivery Risk: 8

Release Overdue 9 days

Batch Size Increase 14 compared to last months

Fig. 4.137: *Use the predictions on the upcoming release as early warnings.*

Pre-Requisites for the Delivery Performance Metrics

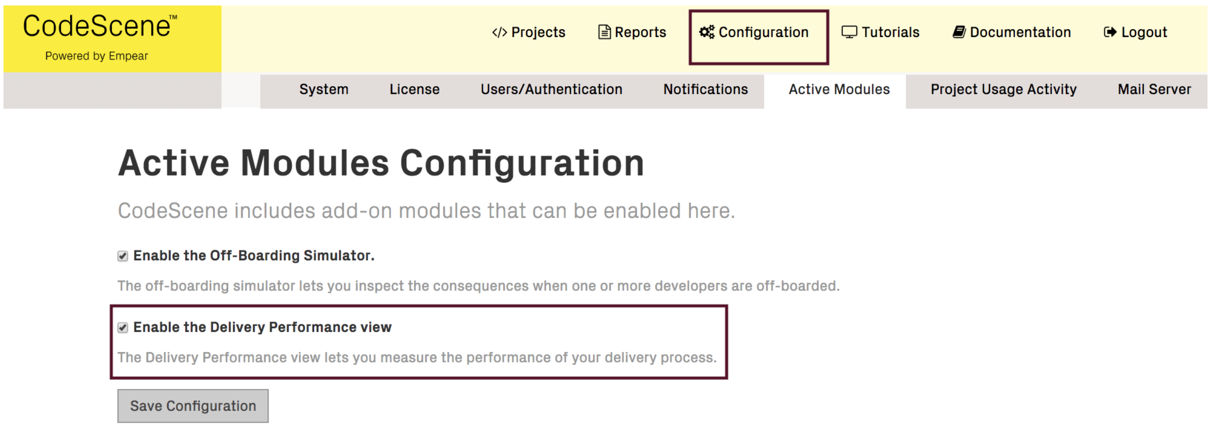
The metrics are based on the following data sources:

- Jira is used to fetch issues and information about the issue type, transitions, and creation dates.
- Commit data is used to calculate the scope, batch sizes, and cycle times.
- Git Tags are used to determine a release.

The Jira data is optional, but without it you cannot get the total lead time or time for bug fixes.

The Git Tags to identify a release are mandatory.

The Delivery Performance module is currently in beta status, which means it has to be enabled by a CodeScene administrator as shown in



Active Modules Configuration

CodeScene includes add-on modules that can be enabled here.

☑ Enable the Off-Boarding Simulator.

The off-boarding simulator lets you inspect the consequences when one or more developers are off-boarded.

☑ Enable the Delivery Performance view

The Delivery Performance view lets you measure the performance of your delivery process.

Fig. 4.138: Enable the Delivery Performance module in the global CodeScene configuration.

4.8 Simulations

4.8.1 Team Planning with the On- and Off-Boarding Simulation

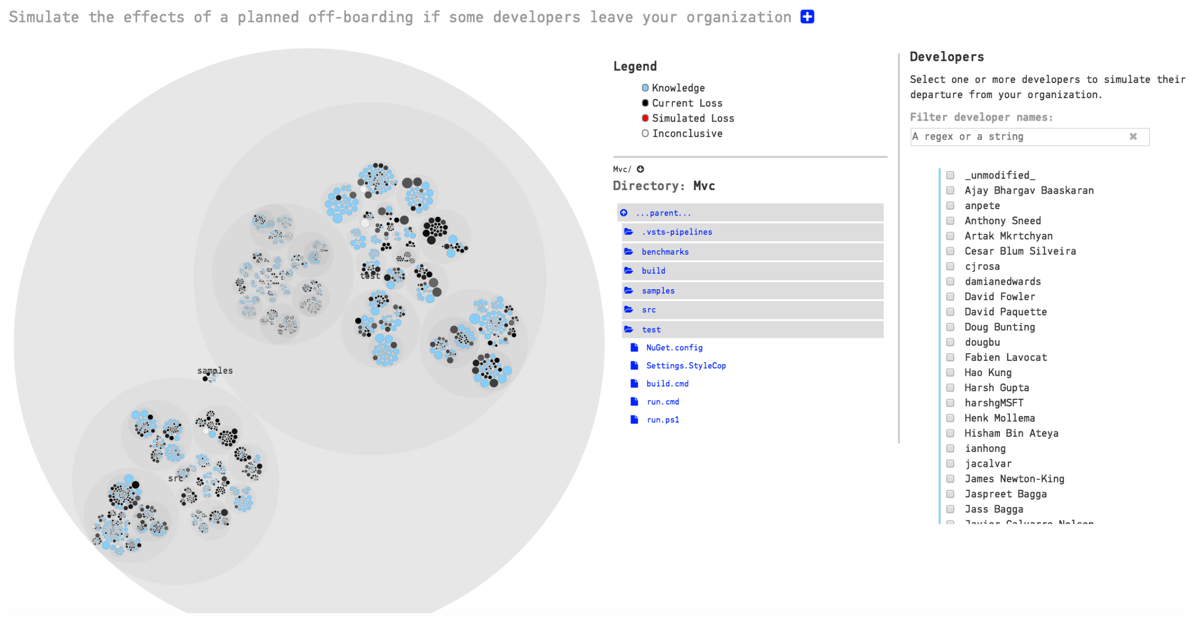
CodeScene's existing knowledge loss analysis provides after the fact information. While that information is useful as input to planning and risk management, it is usually more important to get data we can act upon pro-actively.

For this purpose, CodeScene comes with a simulation module that lets you explore the effects of a planned off-boarding while the developers are still aboard. This gives you the opportunity to identify off-boarding risks and areas of the code in need of a new main developer. Let's see how we use the simulation.

Access the Off-Boarding Simulation

The off-boarding simulation is accessible from each analysis view in a separate module, as shown in Fig. 4.139.

The former contributors in your codebase, the ones configured as ex-developers in your project, are visualized using the black color. The simulation also lists all known developers. To simulate the impact of an off-boarding, select one or more developers from that list as shown in Fig. 4.140. The areas of the code affected by the off-boarding are then highlighted in red. Note that you can search and filter in the developer list, a feature that's useful in projects with many contributors.



Simulate the effects of a planned off-boarding if some developers leave your organization

Legend

- ● Knowledge
- ● Current Loss
- ● Simulated Loss
- ○ Inconclusive

Mvc/

Directory: Mvc

- ...parent...
- .vsts-pipelines
- benchmarks
- build
- samples
- src
- test
- NuGet.config
- Settings.StyleCop
- build.cmd
- run.cmd
- run.ps1

Developers

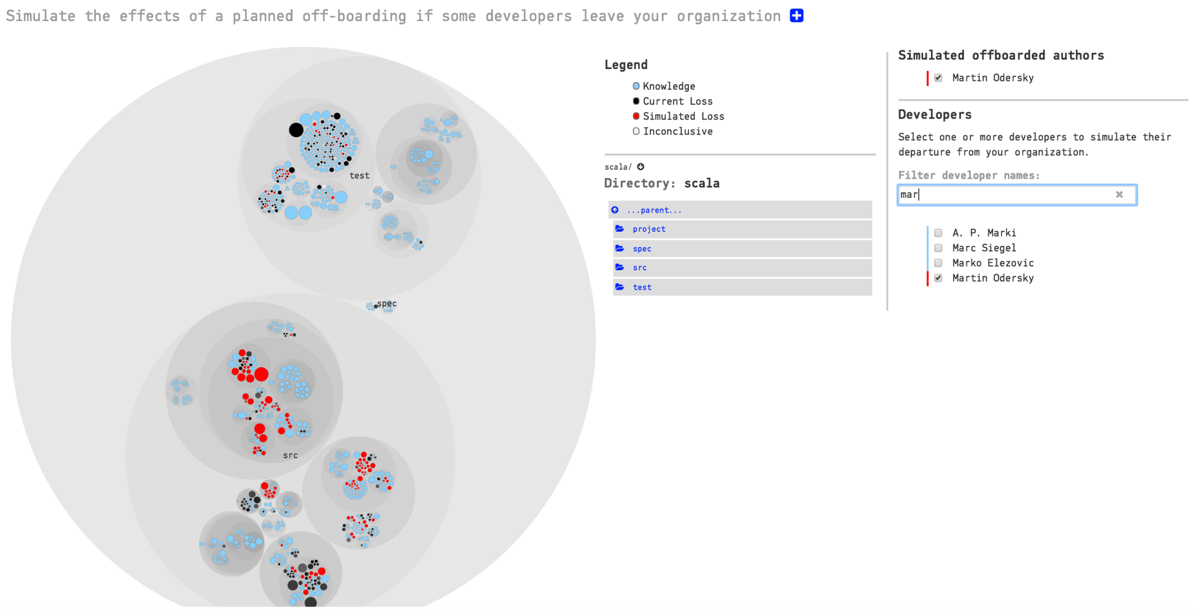
Select one or more developers to simulate their departure from your organization.

Filter developer names:

A regex or a string

- ☐ _unmodified_
- ☐ Ajay Bhargav Baaskaran
- ☐ anpete
- ☐ Anthony Sneed
- ☐ Artak Mkrtchyan
- ☐ Cesar Blum Silveira
- ☐ cjrosa
- ☐ damianedwards
- ☐ David Fowler
- ☐ David Paquette
- ☐ Doug Bunting
- ☐ dougbu
- ☐ Fabien Lavocat
- ☐ Hao Kung
- ☐ Harsh Gupta
- ☐ harshgMSFT
- ☐ Henk Mollema
- ☐ Hisham Bin Ateya
- ☐ ianhong
- ☐ jacalvar
- ☐ James Newton-King
- ☐ Jaspreet Bagga
- ☐ Jass Bagga

Fig. 4.139: Access the off-boarding simulator in the SIMULATIONS module.



Simulate the effects of a planned off-boarding if some developers leave your organization

Legend

- ● Knowledge
- ● Current Loss
- ● Simulated Loss
- ○ Inconclusive

scala/

Directory: scala

- ...parent...
- project
- spec
- src
- test

Simulated offboarded authors

- ☑ Martin Odersky

Developers

Select one or more developers to simulate their departure from your organization.

Filter developer names:

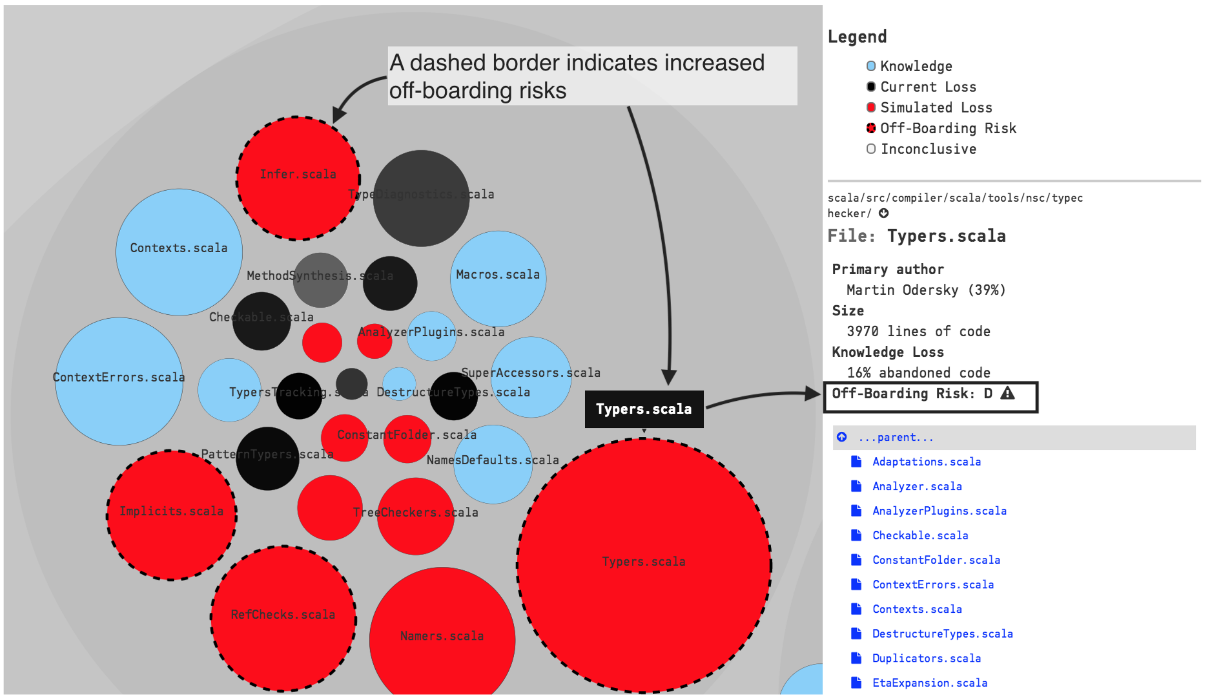
mar

- ☐ A. P. Marki
- ☐ Marc Siegel
- ☐ Marko Elezovic
- ☑ Martin Odersky

Fig. 4.140: Select one or more developers to simulate the off-boarding effect.

Detect Off-Boarding risks

CodeScene auto-detects high risk areas in the off-boarding simulation. That is, if a major hotspot is in the head of a developer who might leave, we consider that an increased off-boarding risk. The off-boarding risks are highlighted as illustrated in Fig. 4.141.



scala/src/compiler/scala/tools/nsc/typechecker/

File: Typers.scala

Primary author
Martin Odersky (39%)
Size
3970 lines of code
Knowledge Loss
16% abandoned code
Off-Boarding Risk: D ⚠

Fig. 4.141: CodeScene auto-detects off-boarding risks.

Act on the Off-Boarding Information

The off-boarding simulation lets you identify the areas of the system where most code has been written by developers who might become former contributors in the future. This might happen for several reasons: a developer or contractor could leave the organization, or maybe a team of developers are re-assigned to a different project.

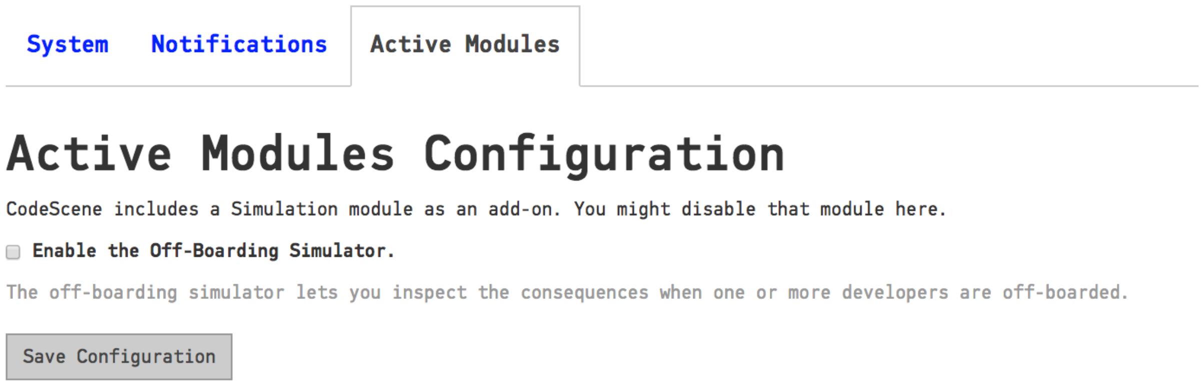
In both situations you typically have a period of time when the soon to be former contributors are still around to support the on-boarding of new people. Using the simulation, you can:

- *Guide on-boarding*: If you find a that a high-risk area gets abandoned, use this information to on-board a new developer in that part of the code.
- *Support planning and priorities*: If the simulation shows that the organization will lose active knowledge of entire components or sub-systems, then you might have to re-prioritize or re-plan features that require extensions of those components. Typically, this means scheduling additional time for learning.
- *Look for upcoming technical gaps*: Some codebases have a high degree of technical sprawl. This means that an off-boarding could lead to a situation where you have code implemented in a programming language that none of the teams master. As such, you want to compare the off-boarding data with CodeScene's analysis of Technical Sprawl. The outcome of this analysis should influence training, hiring, and rewrite decisions.

In particular, look for components that are entirely in the heads of former contributors. That's where the largest risk is.

Disable the Off-Boarding Simulation

The off-boarding simulation isn't intended for performance evaluations. In fact, we strongly advice against that usage and it might even be illegal in certain jurisdictions. With these aspects in mind, you can disable the simulation if you login as an administrator and go to the global configuration in CodeScene as shown in Fig. 4.142.



Active Modules Configuration

CodeScene includes a Simulation module as an add-on. You might disable that module here.

☐ Enable the Off-Boarding Simulator.

The off-boarding simulator lets you inspect the consequences when one or more developers are off-boarded.

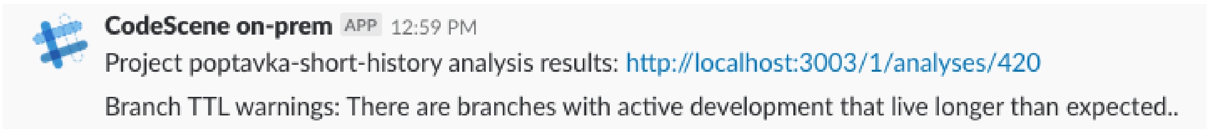
Fig. 4.142: The off-boarding simulation can be disabled in the global configuration.

4.9 Miscellaneous

4.9.1 Notifications

CodeScene comes with a flexible notification system which you can use to receive notification messages when a CodeScene analysis is run.

You can see an example of a Slack notification message in Fig. **??**.



CodeScene on-prem 12:59 PM
Project poptavka-short-history analysis results: http://localhost:3003/1/analyses/420
Branch TTL warnings: There are branches with active development that live longer than expected..

Default notifications start with a project name and the analysis results link. Then comes the list of analysis warnings (if any).

You can customize the content of notification messages in the CodeScene configuration.

Generic Configuration

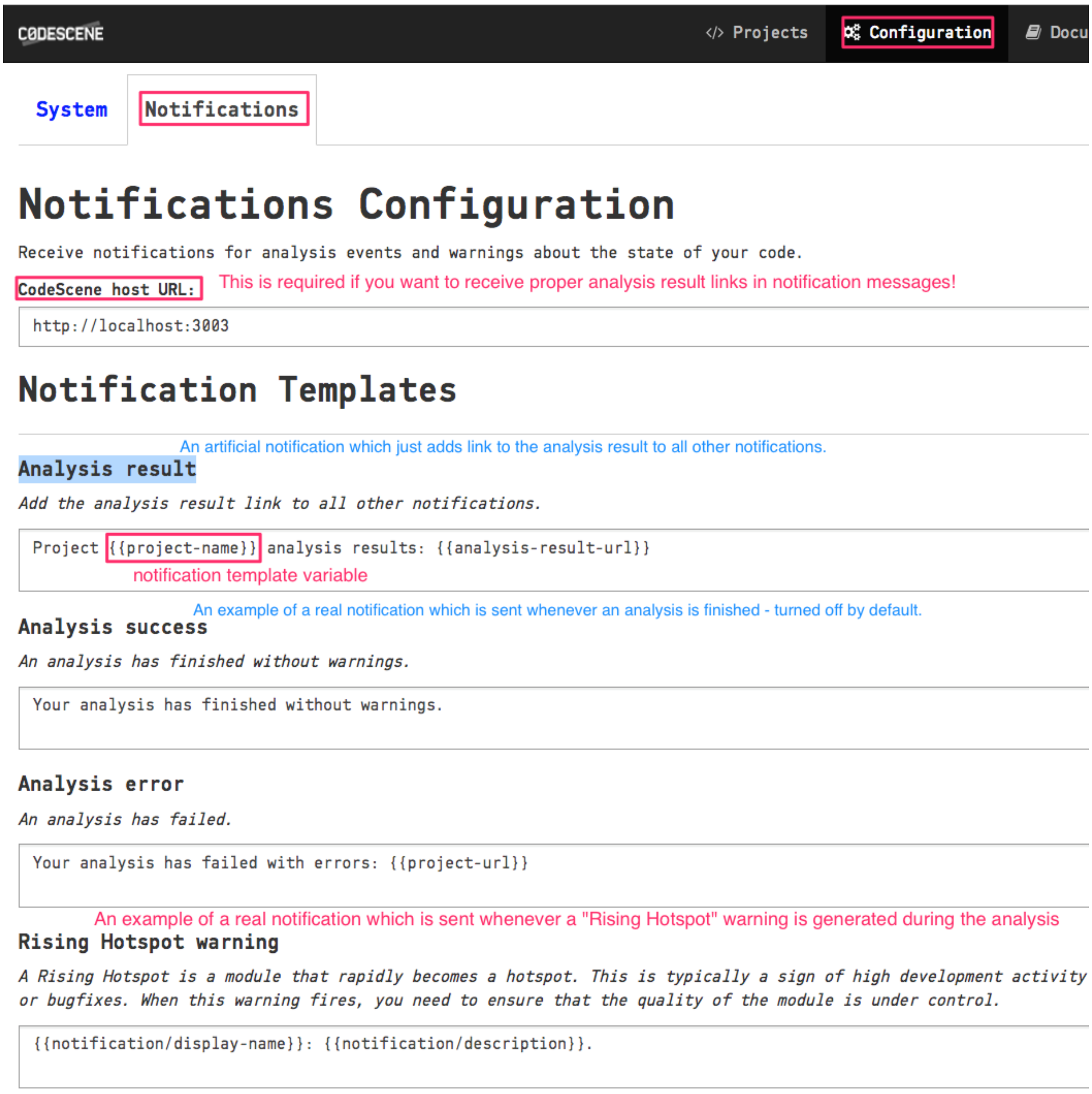
CodeScene administrators can customize notifications in the global configuration on the *Notifications* tab.

Here's an example:

CodeScene host URL must be configured properly if you want to receive proper links to your analyses' results. Make sure that the protocol, the host and the port are all correct.

There are two subsections in the *Notifications Configuration*:

1. *Notification Templates* - generic notification templates for the actual message that will be sent
2. *Slack Notifications* - Slack-specific configuration settings



Notifications Configuration

Receive notifications for analysis events and warnings about the state of your code.

CodeScene host URL: This is required if you want to receive proper analysis result links in notification messages!

http://localhost:3003

Notification Templates

An artificial notification which just adds link to the analysis result to all other notifications.

Analysis result

Add the analysis result link to all other notifications.

```
Project {{project-name}} analysis results: {{analysis-result-url}}
```

notification template variable

An example of a real notification which is sent whenever an analysis is finished - turned off by default.

Analysis success

An analysis has finished without warnings.

```
Your analysis has finished without warnings.
```

Analysis error

An analysis has failed.

```
Your analysis has failed with errors: {{project-url}}
```

An example of a real notification which is sent whenever a "Rising Hotspot" warning is generated during the analysis

Rising Hotspot warning

A Rising Hotspot is a module that rapidly becomes a hotspot. This is typically a sign of high development activity or bugfixes. When this warning fires, you need to ensure that the quality of the module is under control.

```
{{notification/display-name}}: {{notification/description}}.
```

Notification Templates

These are generic notification templates which define the content of notification messages.

You can see a typical example of a notification template in Fig. **??**.

- *Analysis result* is a special type of notification template which is only used to add the analysis result link to all other notifications.
- *Analysis success* is a simple notification that is sent whenever an analysis is run - it's turned off by default to avoid noisy notifications.
- *Analysis error* is only sent when an analysis fails with an unrecoverable error.
- *Rising Hotspot warning* is an example of a notification which is sent when your analysis finishes with "Rising Hotspot" warning.

Template Variables

You can use several variables in your notification templates. You use them by wrapping them with double curly braces, e.g. *{{notification/display-name}}*.

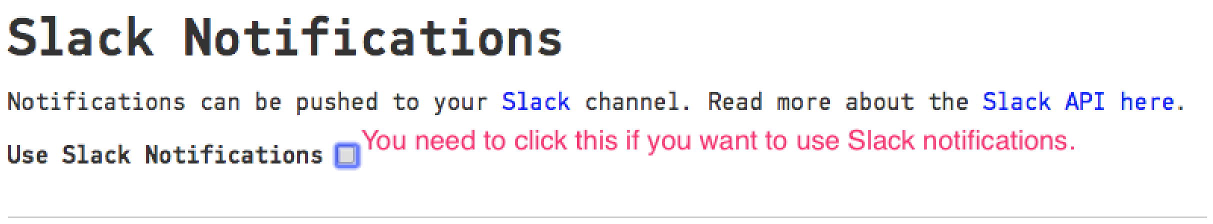
The following variables are available:

- *project-name* - the name of the project
- *project-url* - an absolute URL to the project details; can be useful when the analysis fails
- *analysis-result-url* - an absolute URL to the project analysis results
- *notification/display-name* - a display name of the notification, e.g. *Rising Hotpost warning*; it's mostly useful for warnings and corresponds to the labels used for notification templates in configuration.
- *notification/description* - a short description of each notification; again, mostly useful for warnings and visible under the notification labels in configuration.

Slack Notifications

Slack Notifications configuration are separate from the generic notification settings.

They are hidden by default so you need to enable them first:

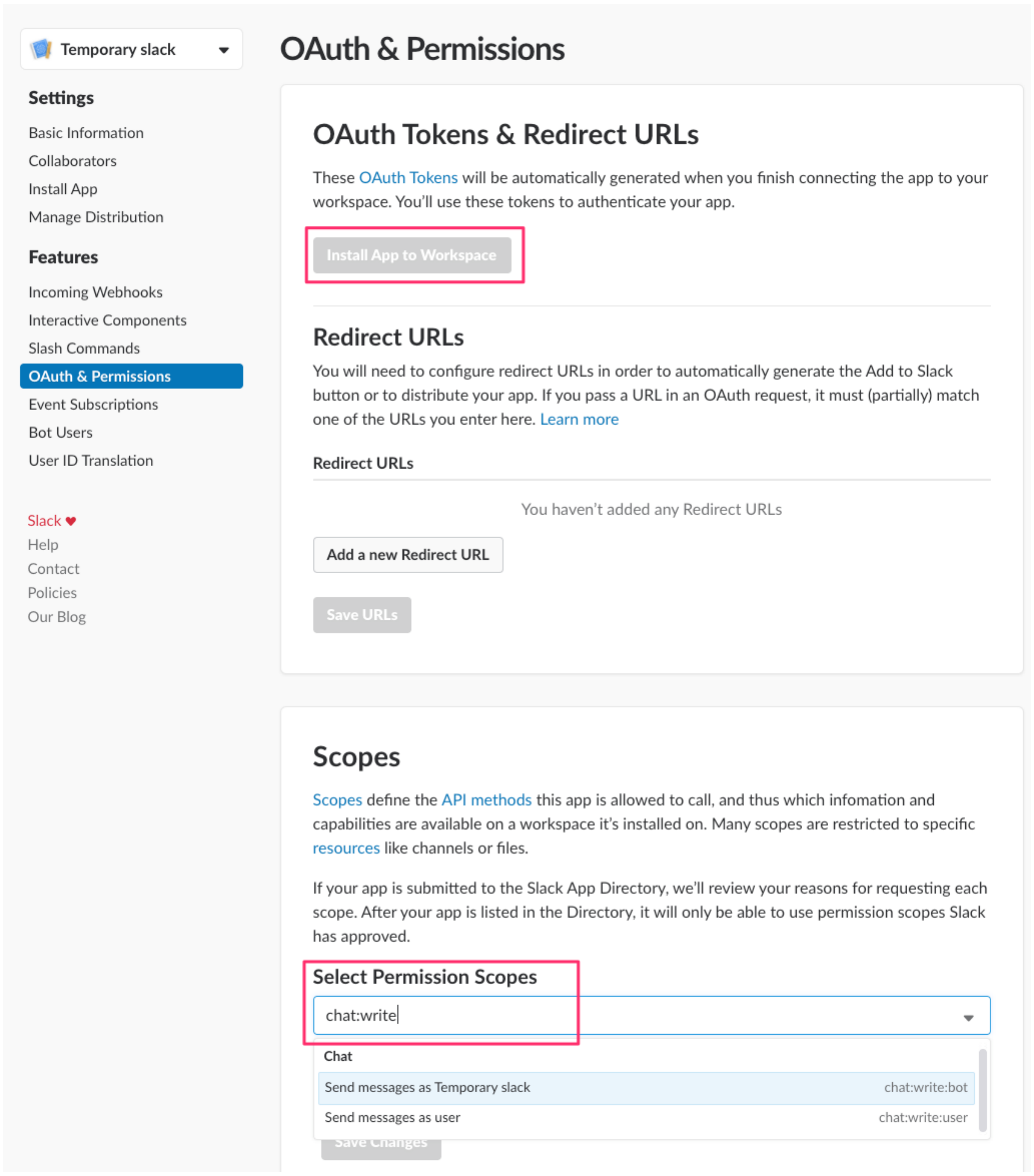

Slack Notifications
Notifications can be pushed to your Slack channel. Read more about the Slack API here.
Use Slack Notifications
You need to click this if you want to use Slack notifications.

Slack Application Setup

Before starting, you need to set up your Slack application. Click the *Create a Slack app* button and fill in the details. When the application is created, select *OAuth & Permissions* in the left panel and pick the *chat:write:bot* permission in the *Select Permissions Scopes* combo box. Save your changes and then click the *Install App to Workspace* button at the top of the page:

After the Slack application is installed you should be able to get your access token:

Copy and paste this token into the CodeScene Slack configuration.



OAuth & Permissions

OAuth Tokens & Redirect URLs

These OAuth Tokens will be automatically generated when you finish connecting the app to your workspace. You'll use these tokens to authenticate your app.

Redirect URLs

You will need to configure redirect URLs in order to automatically generate the Add to Slack button or to distribute your app. If you pass a URL in an OAuth request, it must (partially) match one of the URLs you enter here. Learn more

Redirect URLs

You haven't added any Redirect URLs

Scopes

Scopes define the API methods this app is allowed to call, and thus which infomation and capabilities are available on a workspace it's installed on. Many scopes are restricted to specific resources like channels or files.

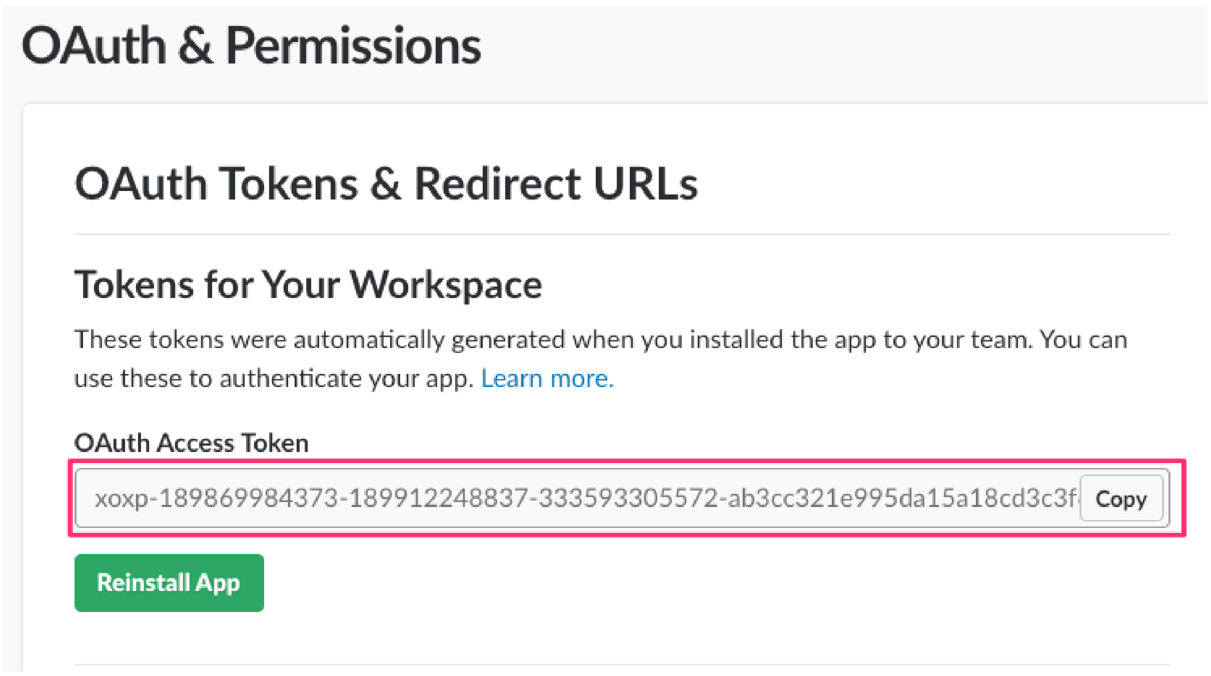
If your app is submitted to the Slack App Directory, we'll review your reasons for requesting each scope. After your app is listed in the Directory, it will only be able to use permission scopes Slack has approved.

Select Permission Scopes

chat:write

Chat

Send messages as Temporary slack	chat:write:bot
Send messages as user	chat:write:user


OAuth & Permissions

OAuth Tokens & Redirect URLs

Tokens for Your Workspace

These tokens were automatically generated when you installed the app to your team. You can use these to authenticate your app. Learn more.

OAuth Access Token

xoxp-189869984373-189912248837-333593305572-ab3cc321e995da15a18cd3c3f

Slack Notifications Config

Slack Notifications configuration comes down to the two important things:

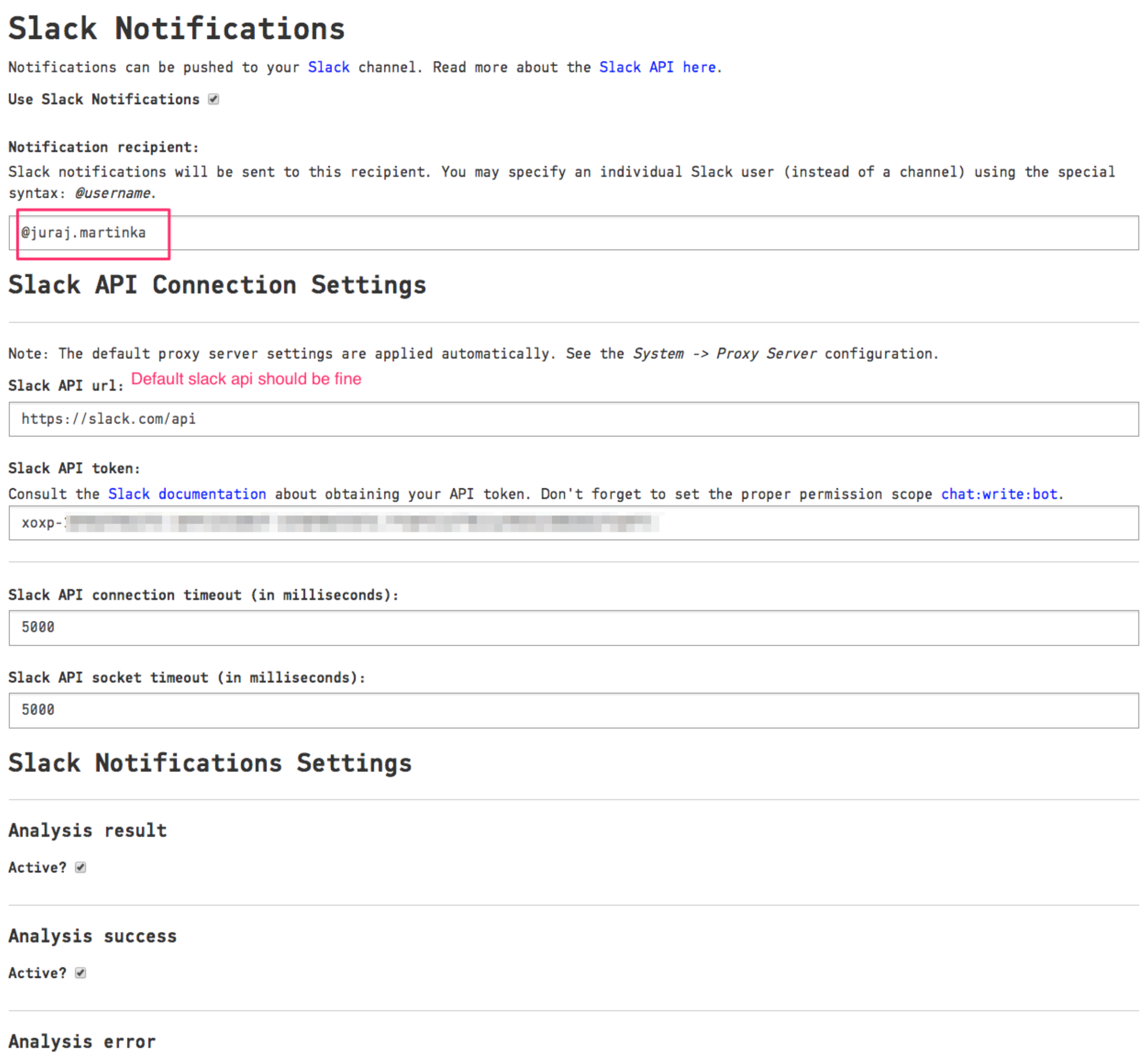
- *Notification recipient* - the name of the channel (of the user) where the notifications should be sent
- *Slack API token* - the access token created in the previous step

Note: for *Notification recipient*, use the name of the Slack channel, as is, e.g. *codescene-slack*. If you want to send notifications to the specific user, then prepend @ to the username, .g. *@juraj.martinka* for sending notifications to the slack user *juraj.martinka*.

You can see a sample configuration in Fig. **??**.

Once you have everything configured click *Save Slack Notifications Settings*.

You are ready to run an analysis and recieve your first notification!



Slack Notifications

Notifications can be pushed to your Slack channel. Read more about the Slack API here.

Use Slack Notifications ✓

Notification recipient:
Slack notifications will be sent to this recipient. You may specify an individual Slack user (instead of a channel) using the special syntax: *@username*.

@juraj.martinka

Slack API Connection Settings

Note: The default proxy server settings are applied automatically. See the *System -> Proxy Server* configuration.

Slack API url: Default slack api should be fine

https://slack.com/api

Slack API token:
Consult the Slack documentation about obtaining your API token. Don't forget to set the proper permission scope chat:write:bot.

xoxp-[redacted]

Slack API connection timeout (in milliseconds):

5000

Slack API socket timeout (in milliseconds):

5000

Slack Notifications Settings

Analysis result

Active? ✓

Analysis success

Active? ✓

Analysis error

Chapter 5

Configuration

5.1 Project Configuration

5.1.1 Specify the Git Repository to Analyze

Your first step is to tell CodeScene where your code is. There are six 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. 5.1. 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.

4. Clone an existing analysis configuration. CodeScene copies all your configuration options – filters, repository paths, exclusions, teams, ex-developer configuration, etc – to a new project. From here those two projects (the original and the clone) are completely independent and changes to one of them do not affect the other.

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

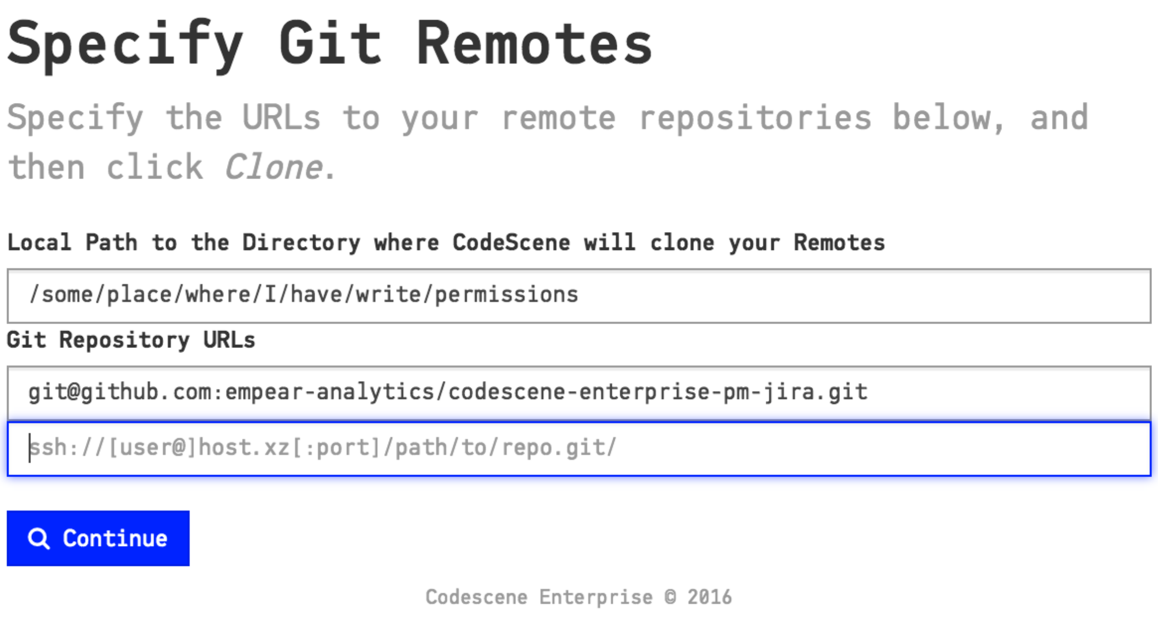
5. Use Google's Repo tool. You provide CodeScene with the URL to the repository containing your project's manifest file. CodeScene will then initialize a local directory as a Repo project and clone all of your Git repositories.

With a Repo-based project, you can switch between branches of the manifest to check out different versions of your project. Branch selection inside the project's Git repositories can be controlled through the manifest file.

6. Import a project configuration that has been created by exporting the configuration of a project from this or another CodeScene instance.

5.1.2 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.



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

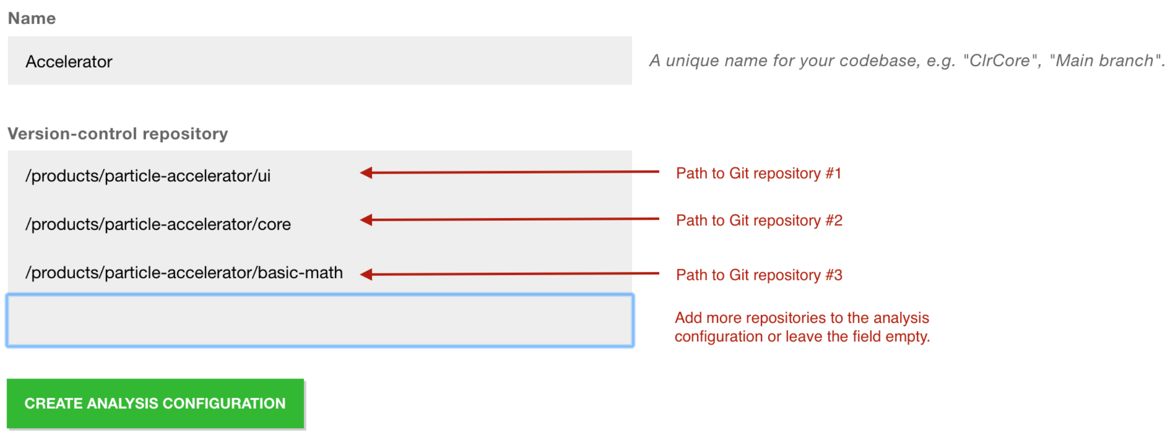
git@github.com:empear-analytics/codescene-enterprise-pm-jira.git

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

Fig. 5.1: Let CodeScene clone your Git repositories through their URL.

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:



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. 5.2: Configuration of multiple repositories.

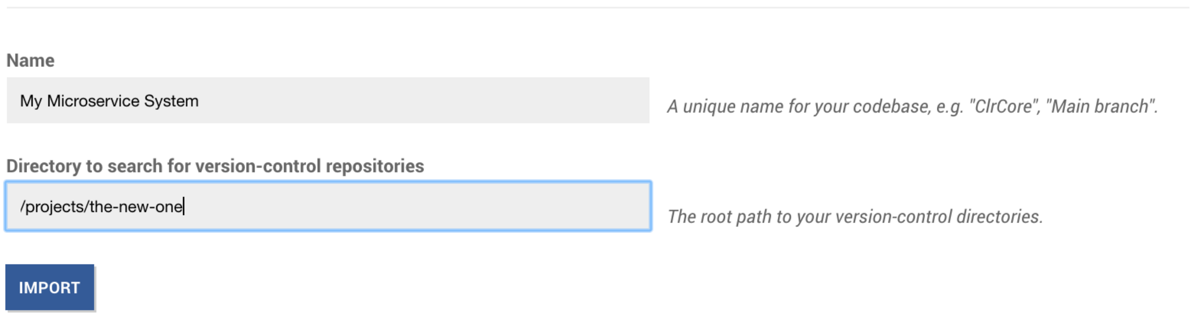
The screenshot above shows three repositories that belong to the same product. During an analysis, CodeScene will analyze the evolution of the code in all those repositories *as though they were in the same physical Git repository.*

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.

5.1.3 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".

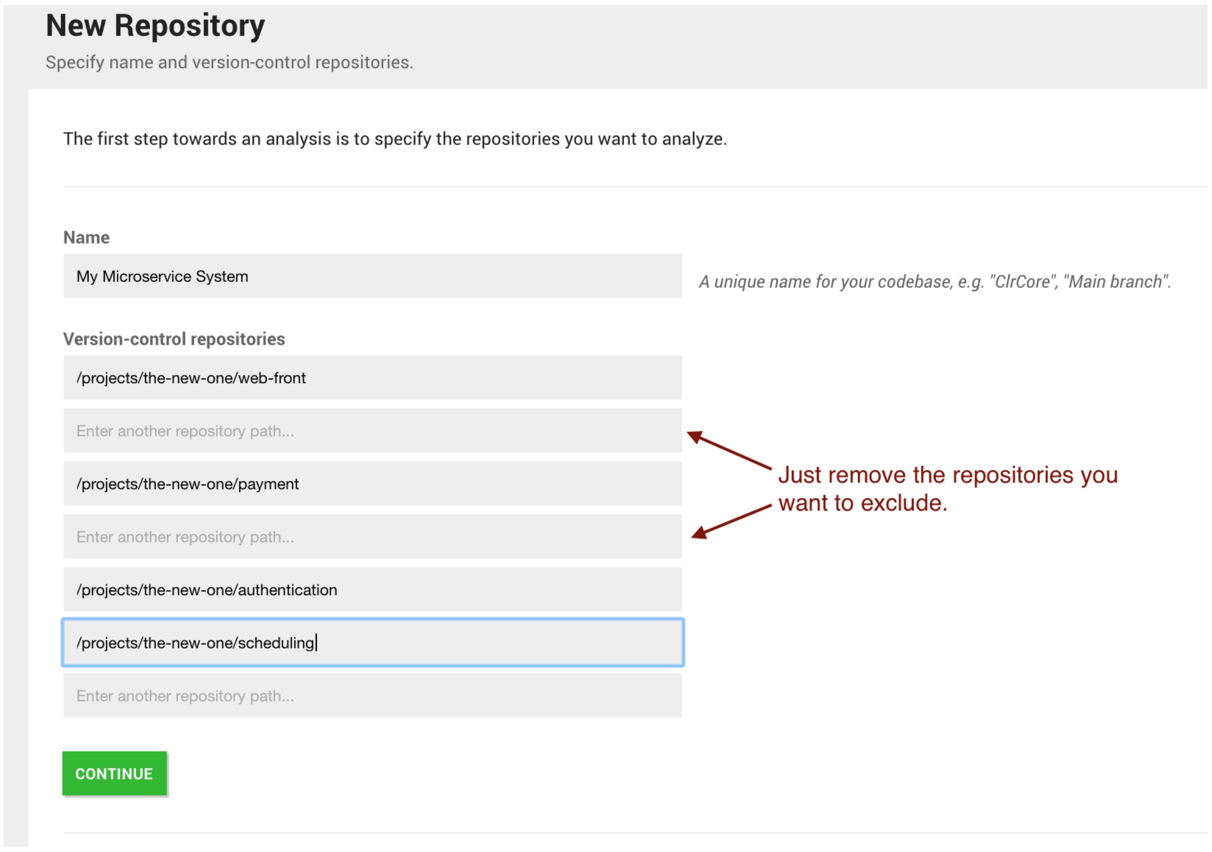
Directory to search for version-control repositories

/projects/the-new-one

The root path to your version-control directories.

Fig. 5.3: 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...

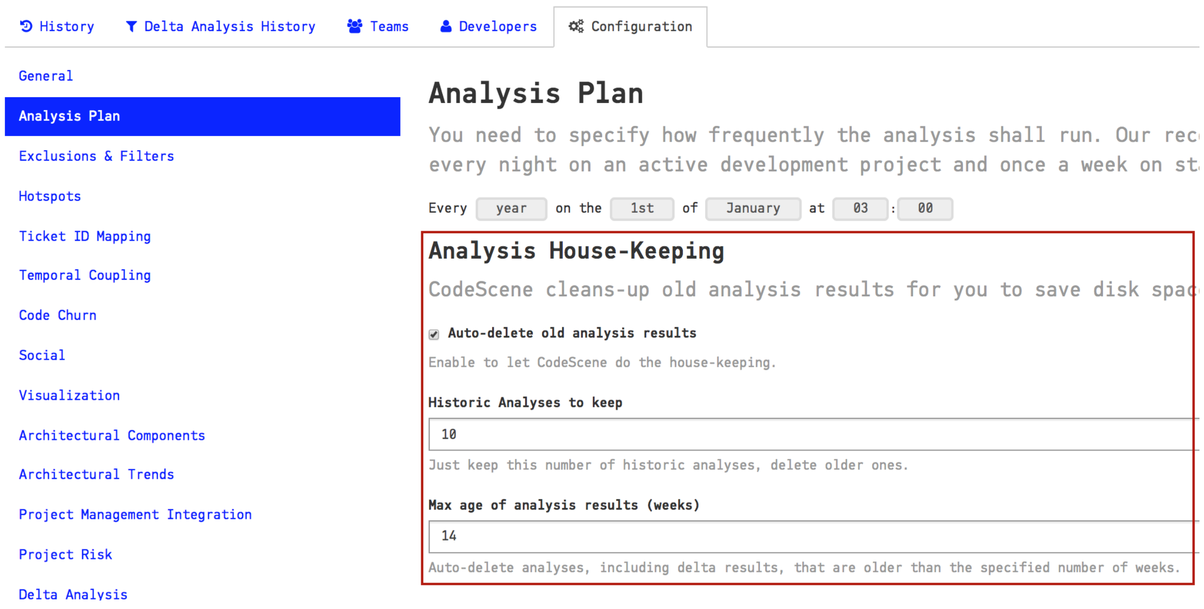
Fig. 5.4: The result of auto importing multiple repositories.

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.

5.1.4 Tune the House-Keeping Options for Analysis Results

CodeScene is designed to run continuously to monitor your system. That also means you will accumulate lots of historic analysis results that occupy space on your host machine.

CodeScene lets you specify a house-keeping strategy that automatically cleans out old historic results, as illustrated in Fig. 5.5.



Analysis Plan

You need to specify how frequently the analysis shall run. Our rec
every night on an active development project and once a week on st

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

Analysis House-Keeping

CodeScene cleans-up old analysis results for you to save disk spac

✓ Auto-delete old analysis results

Enable to let CodeScene do the house-keeping.

Historic Analyses to keep

10

Just keep this number of historic analyses, delete older ones.

Max age of analysis results (weeks)

14

Auto-delete analyses, including delta results, that are older than the specified number of weeks.

Fig. 5.5: Specify how much history you want to keep.

5.1.5 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:

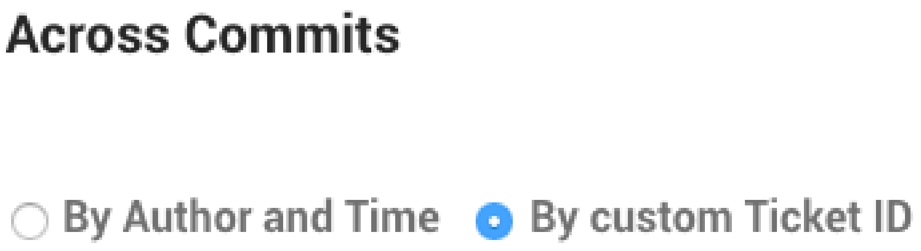
By Author and Time
: When you specify this option, the tool will consider all commits by the same author on the same day as a single, logical commit. This option is a heuristic that works well in the absence of a Ticket ID in your data.

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. 5.6 shows the options in the repository configuration section *Temporal Coupling*.

To aggregate by custom Ticket ID, you need specify a *Ticket ID Pattern*, in the *Ticket ID Mapping* section (see Fig. 5.7). The pattern is used to extract the Ticket ID from the commit message. The example pattern in Fig. 5.7 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.

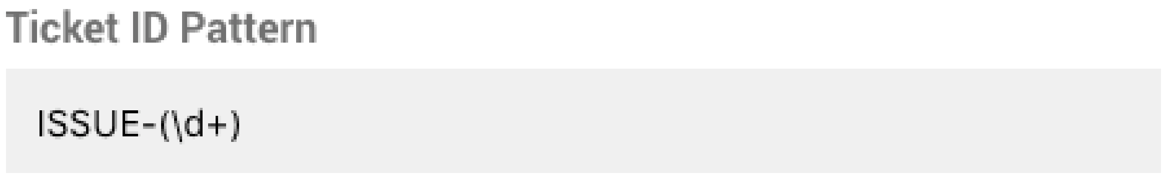
Note that CodeScene will still calculate normal temporal coupling on a single commit basis. You want that in order to spot unexpected dependencies between files in the same repository. The temporal coupling results for the logical commits discussed above are presented in a separate analysis view.



Across Commits

By Author and Time [✓ By custom Ticket ID]

Fig. 5.6: There are two available strategies for aggregating commits.



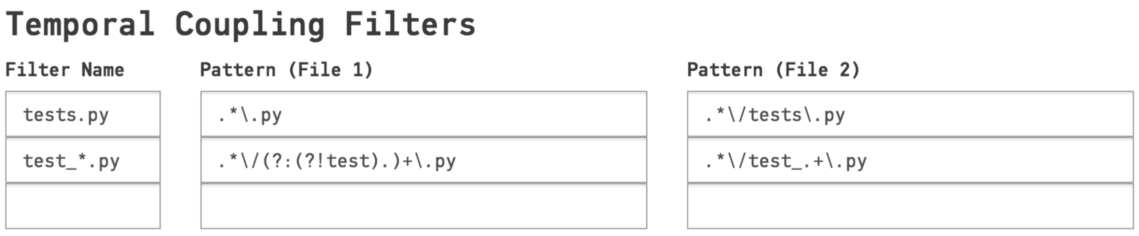
Ticket ID Pattern

ISSUE-(\d+)

Fig. 5.7: Configure a pattern to extract a Ticket ID.

5.1.6 Temporal Coupling Exclusion Filters

You might have files that you expect to be temporally coupled, for example tests and the corresponding units under test, or matching *.c* and *.h* files. To exclude these coupling from visualization by default, go to the "Temporal Coupling" section of the project configuration and add "Temporal Coupling Filters" for the patterns you want to exclude, as shown in Fig. 5.8.



Temporal Coupling Filters

Filter Name	Pattern (File 1)	Pattern (File 2)
tests.py	`.*\.py`	`.*\/tests\.py`
test_*.py	`.*\/(?:(?!test).)+\.py`	`.*\/test_.+\.py`

Fig. 5.8: Configure temporal coupling filters for expected file couplings.

Each filter has a *name*, that can be anything you like, and *patterns* for coupled file paths. The patterns are a regular expressions. When a pair of coupled files match the patterns, in either direction, they are excluded by the filter.

All filters are tried in sequence, and if any filter hits a coupled pair, the pair is excluded. Some useful examples of patterns are:

Pattern (File 1)	Pattern (File 2)	Description
`.+\.(?:c|cc|cpp|cxx)`	`.+\.(?:h|hh|hxx)`	C/C++ includes, e.g. `gc.cpp` and `util.h`
`.+\/(.+)\.java`	`.+\/(.+)Impl\.java`	Java "Impl" pairs, e.g. `Thing.java` and `ThingImpl.java`
`.+\/(.+)\.cs`	`.+\/I(.+)\.cs`	C# interface pairs, e.g. `IComponent.cs` and `Component.cs`
`.*\/(?:(?!test).)+\.py`	`.*\/test_.+\.py`	Python files and tests, e.g. `foo/a.py` and `tests/test_a.py`

If any of the patterns have capturing groups, both matches must generate the same number of captures, with equal values, to trigger the filter. Note that non-capturing groups and negative look-ahead in regular expressions can be useful if you want to write advanced filters, and only trigger filters on corresponding files in corresponding directories.

5.1.7 Linking to an External Ticket System

If you have a Ticket ID Pattern configured, and a way to deep-link to tickets by the matched identifiers, you can configure a *Ticket URI Template* to enable links in analysis views. That way you will be able to quickly navigate from Code Churn by Task to the external ticket system, and view more details there.

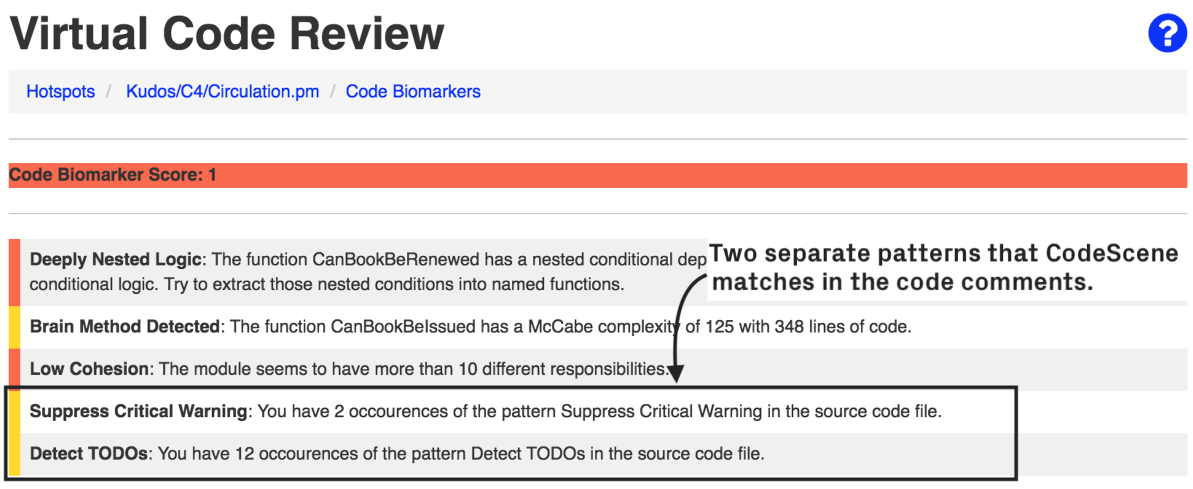
The Ticket URI Template is based on the URI Template format (RFC 6570), with support for the single expression `{ticket-id}`. The matched ticket value, i.e. the captured value of the regular expression group, is used as `{ticket-id}` for hyperlinks. For example, if your Ticket ID Pattern is `#(\d+)`, and your Ticket URI Template is `https://example.com/tickets/{ticket-id}`, a commit containing the string `#1234` will result in a hyperlink to `https://example.com/tickets/1234`.

Some useful examples of Ticket ID Patterns and Ticket Template URIs are:

- **GitHub:** `#(\d+)` and `https://github.com/your-org/your-project/issues/{ticket-id}`
- **JIRA:** `(\[A-Z]{2,}-\d+)` and `https://example.com/jira/browse/{ticket-id}`
- **Trello (Card Numbers):** `CARD-(\d+)` and `https://trello.com/search?q={ticket-id}`
- **Trello (Card Short IDs):** `CARD-(.+)` and `https://trello.com/c/{ticket-id}`

5.1.8 Detect Patterns in Code Comments

Exhaustive use of certain code comments indicate code smells. For example, a file that is filled with *TODO* comments is usually not that reassuring. On a similar notes, organizations might use static analysis tools and use code comments to suppress the findings. By configuring a set of patterns, you can use CodeScene's virtual code reviewer to detect such patterns as shown in Fig. 5.9.


Virtual Code Review
Hotspots / Kudos/C4/Circulation.pm / Code Biomarkers
Two separate patterns that CodeScene matches in the code comments.

Code Biomarker Score: 1

Deeply Nested Logic: The function CanBookBeRenewed has a nested conditional dep conditional logic. Try to extract those nested conditions into named functions.

Brain Method Detected: The function CanBookBeIssued has a McCabe complexity of 125 with 348 lines of code.

Low Cohesion: The module seems to have more than 10 different responsibilities.

Suppress Critical Warning: You have 2 occourences of the pattern Suppress Critical Warning in the source code file.

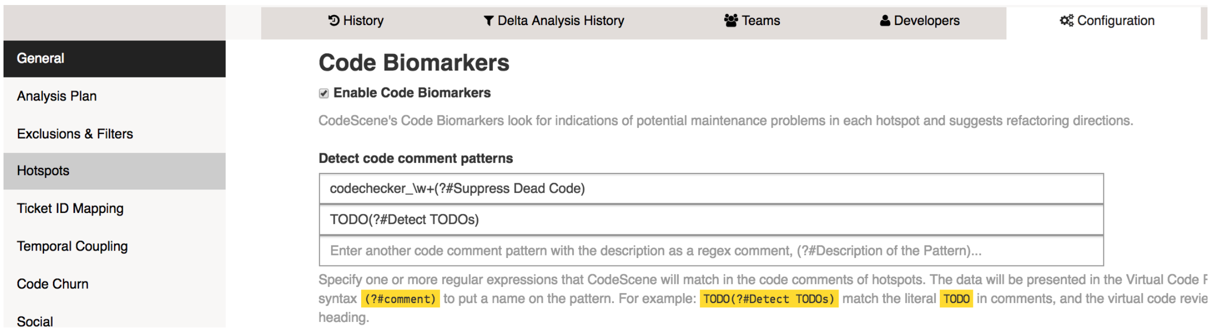
Detect TODOs: You have 12 occourences of the pattern Detect TODOs in the source code file.

Fig. 5.9: Detect specific type of code comments.

The configuration is a bit special, but read along for examples – it's not hard:

Fig. 5.10 presents two patterns that CodeScene will match in the code comments of your hotspots. Each pattern consists of two parts, separated by the regex inline comment syntax, *(?#comment)*:

1. A regular expression to match in the code comments.
2. A descriptive name of the content that the regular expression matches. This will be used in the virtual code reviewer.



Code Biomarkers

☑ Enable Code Biomarkers

CodeScene's Code Biomarkers look for indications of potential maintenance problems in each hotspot and suggests refactoring directions.

Detect code comment patterns

codechecker_\w+(?#Suppress Dead Code)

TODO(?#Detect TODOs)

Enter another code comment pattern with the description as a regex comment, (?#Description of the Pattern)...

Specify one or more regular expressions that CodeScene will match in the code comments of hotspots. The data will be presented in the Virtual Code F syntax `(?#comment)` to put a name on the pattern. For example: `TODO(?#Detect TODOs)` match the literal `TODO` in comments, and the virtual code revie heading.

Fig. 5.10: Configure regular expressions to detect code comments.

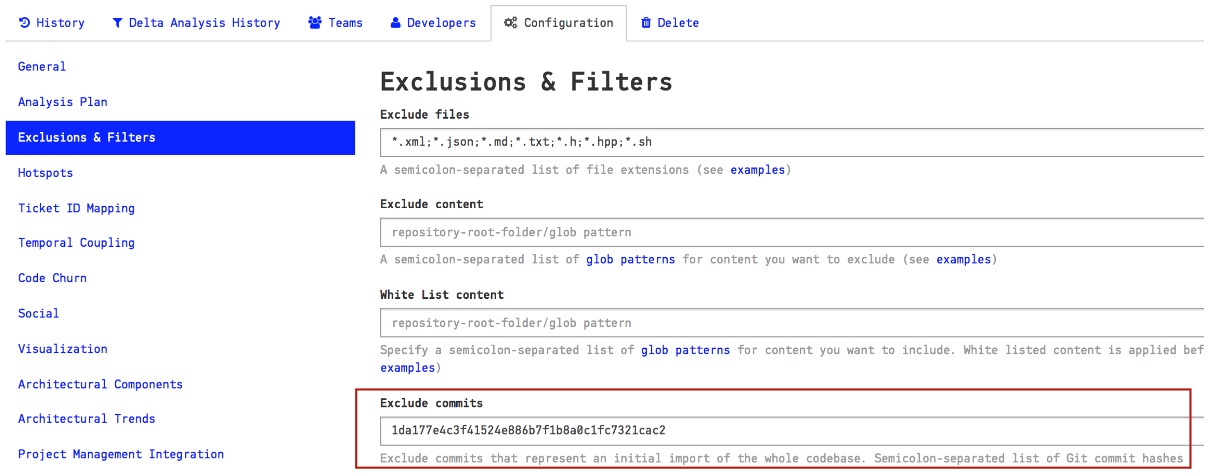
In the first example, we match the expression *codechecker_w+*. That is, any code comment that starts with *codechecker_* followed by a string such as *_confirmed* or *_critical*. We then add the descriptive comment *(?#Suppress Dead Code)*. Note that only "Suppress Dead Code" makes up the name; the *(?#...)* syntax is only to embed the name in the regex.

The second example shows a simpler pattern where we match the literal string *TODO* in a code comment, and associate it with the label "Detect TODOs" which will then be displayed in the virtual code review.

5.1.9 Exclude Initial Commits from an Analysis

Some Git repositories start their life as an import of an existing codebase. If the previous history isn't migrated together with the code, the author that does the initial commit of the existing codebase gets all the credit. This leads to a bias in the social analyses.

The solution is to exclude all contributions done as part of the initial commit. You specify those commits (fetch them from your Git log) in the project configuration as shown in Fig. 5.11.



Exclusions & Filters

Exclude files

.xml;.json;*.md;*.txt;*.h;*.hpp;*.sh

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 to exclude (see examples)

White List content

repository-root-folder/glob pattern

Specify a semicolon-separated list of glob patterns for content you want to include. White listed content is applied bef examples)

Exclude commits

1da177e4c3f41524e886b7f1b8a0c1fc7321cac2

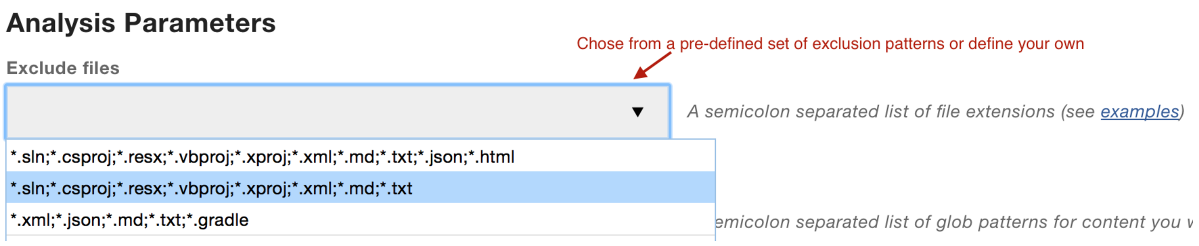
Exclude commits that represent an initial import of the whole codebase. Semicolon-separated list of Git commit hashes

Fig. 5.11: Exclude specific commits from the analysis.

5.1.10 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

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

Chose from a pre-defined set of exclusion patterns or define your own
A semicolon separated list of file extensions (see examples)
emicolon separated list of glob patterns for content you v

Fig. 5.12: 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.

5.1.11 Exclude Specific Files and Folders from an Analysis

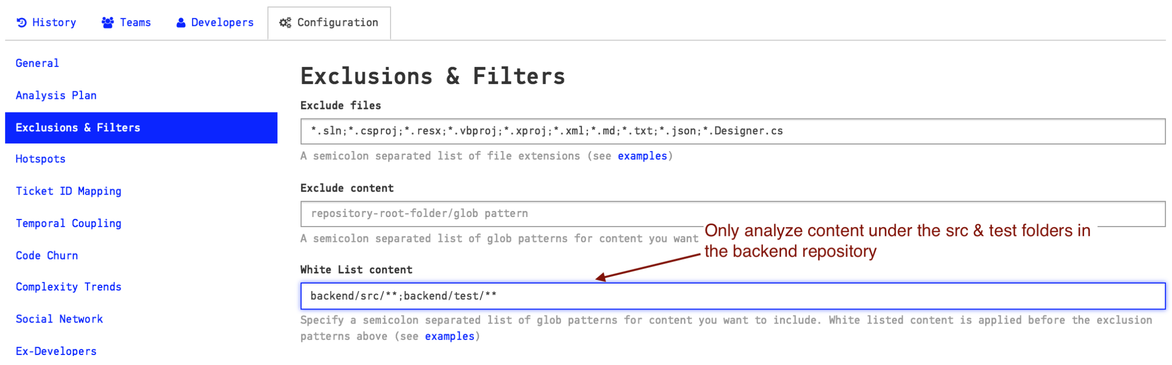
You just learned how you can exclude certain types of files, no matter where they are located in the your codebase. But sometimes you'd like to exclude a particular file or, more often, a complete folder. For example, let's say that you check-in third party code in your repository. You don't want that 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. 5.13.



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

White List content

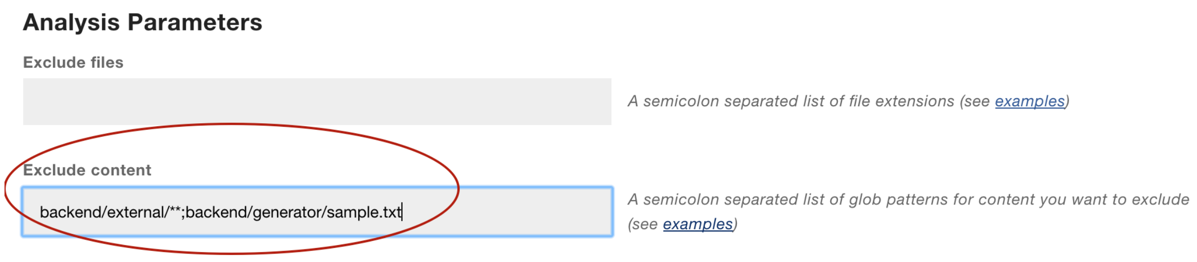
```
backend/src/**;backend/test/**
```

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)

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

Fig. 5.13: Glob patterns to white list content.

You specify a *glob pattern* to Exclude Content from the analysis as illustrated in Fig. 5.14.



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. 5.14: 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.

5.1.12 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/***.

5.1.13 Specify An Analysis Period

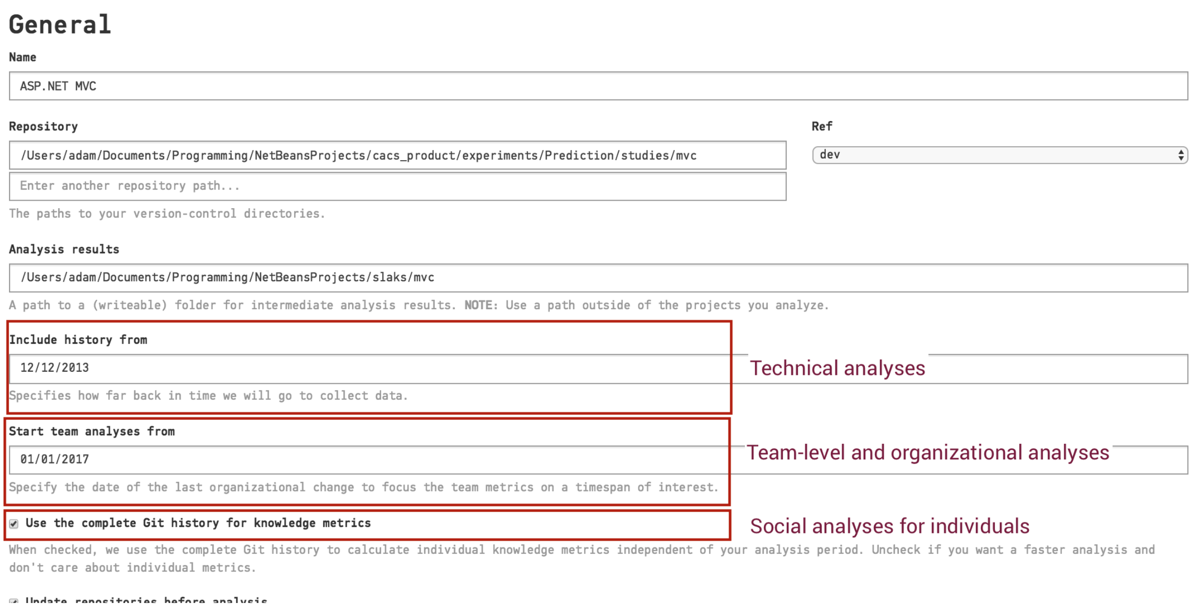
CodeScene lets you specify an *analysis period* as illustrated in Fig. 5.15. That is, how much of your repository history do you want to analyze?

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 uses three different analysis periods depending on the type of information it analyses:

- Technical information and Team information uses the specified start date.


General

Name

ASP.NET MVC

Repository

/Users/adam/Documents/Programming/NetBeansProjects/cacs_product/experiments/Prediction/studies/mvc

Enter another repository path...

The paths to your version-control directories.

Ref

dev

Analysis results

/Users/adam/Documents/Programming/NetBeansProjects/slaks/mvc

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

Include history from

12/12/2013

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

Start team analyses from

01/01/2017

Specify the date of the last organizational change to focus the team metrics on a timespan of interest.

☑ Use the complete Git history for 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.

Technical analyses
Team-level and organizational analyses
Social analyses for individuals

Fig. 5.15: Specify how far back in time you want CodeScene to analyse.

- Configure the team-level analyses to use the date of the last organizational change.
- 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. 5.15).

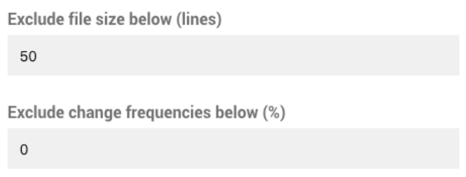
Similar, team-level analyses like coordination needs and Conway's Law should ignore the historic activity of previous organizational structures, and you want to measure from the date where the current team structure got operational.

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.

5.1.14 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. 5.16.


Exclude file size below (lines)

50

Exclude change frequencies below (%)

0

Fig. 5.16: 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.

5.1.15 Working with Repo

The Repo tool is often used for very large projects containing many separate but related Git repositories. A central Manifest XML file is then maintained to define the list of included projects.

CodeScene's Repo integration makes it easier to analyze this kind of large project because you no longer need to enter each sub-project separately. Just point CodeScene at your manifest repository and CodeScene will use Repo to download your code. As your project evolves, CodeScene will keep your analyses in sync, adding and removing Git repositories as necessary.

Using Repo introduces several important differences in how CodeScene works.

Overall approach

When using Repo with CodeScene, your project is controlled through the manifest file. CodeScene synchronizes your project before every analysis, so any changes to your manifest are automatically and immediately taken into account.

CodeScene supports branch selection in your manifest repository. You can select different branches to checkout different versions of your project.

Creating a project with Repo

Repo must be installed on your local machine. If necessary, you can indicate the name of the Repo executable in the CodeScene configuration.

To create your project, go to the "New Project" page, and choose "Google Repo". You will be presented with the following options:

Local path indicates where the new Repo directory will be installed. If the directory does not exist, CodeScene will try to create it.

Repo URL is the URL of your repo manifest Git repository. This value will be used in calls to `repo init -u <URL>` and should be in the format indicated:

```
git@github.com/myorg/my-manifests.git
```

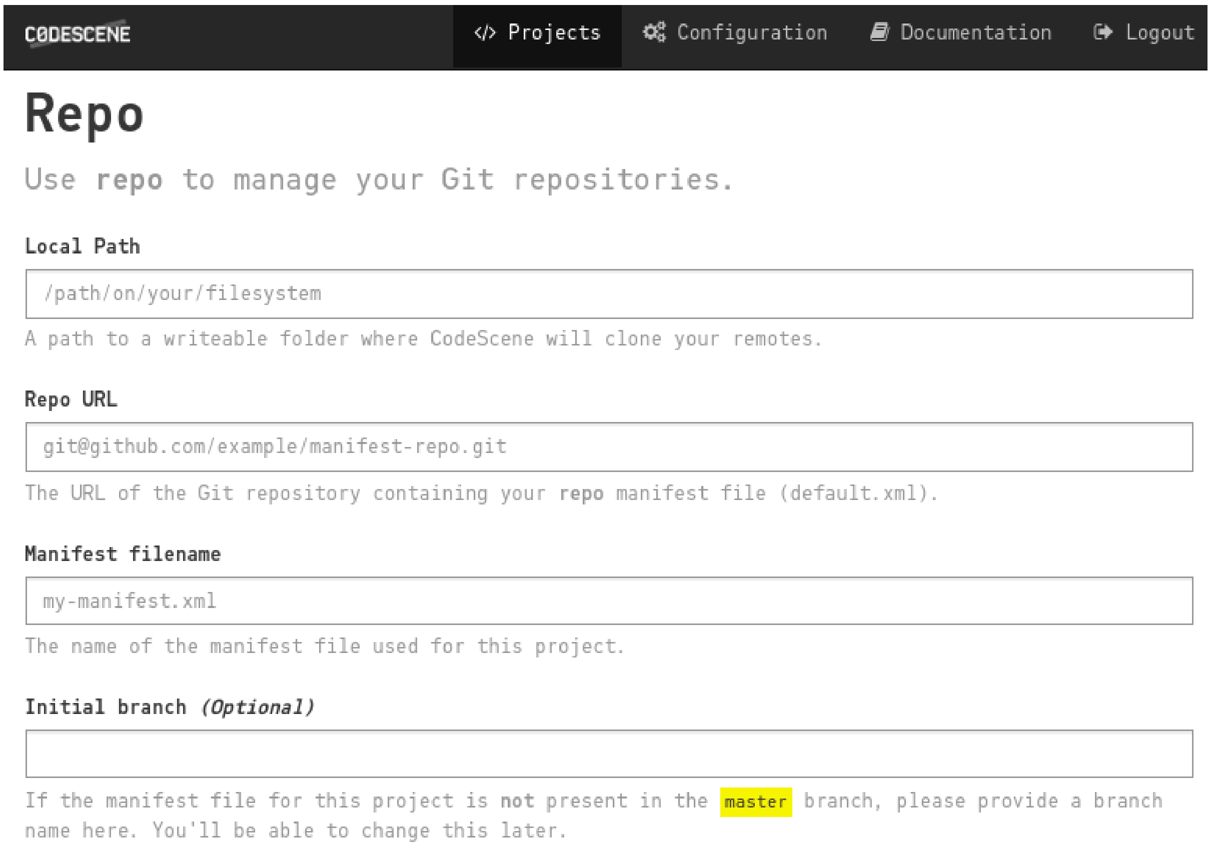
Note that this value cannot be changed later. To change to a new manifest repository, you'll need to create a new project.

Manifest filename is the name of the manifest you'll use. This field is required even if your manifest is *default.xml*. Like the Repo URL, this cannot be changed without creating a new project.

Initial branch only needs to be filled out if the manifest file you wish to use is not available in the *master* branch of your manifest repository. This allows CodeScene to "see" your manifest in order to initialize your project.

When you click on "Initialize", CodeScene will set up the Repo directory and download your manifest file. The next page allows you to check that the Git repos to be clone are correct, and to switch branches if necessary.

CodeScene will then clone your repositories. This may take a long time. When this step is complete, project creation follows the usual path.



Repo

Use **repo** to manage your Git repositories.

Local Path

/path/on/your/filesystem

A path to a writeable folder where CodeScene will clone your remotes.

Repo URL

git@github.com/example/manifest-repo.git

The URL of the Git repository containing your **repo** manifest file (default.xml).

Manifest filename

my-manifest.xml

The name of the manifest file used for this project.

Initial branch ***(Optional)***

If the manifest file for this project is **not** present in the `master` branch, please provide a branch name here. You'll be able to change this later.

Fig. 5.17: Getting started with Repo

Working with Repo-based projects

The primary difference with Repo-based projects is that things like repository selection and branching within Git repositories are handled through the manifest file, either by modifying it in your manifest repository or by switching between branches in CodeScene.

To analyze a specific state of your project, you can use either a branch specification in your manifest file

```
<project name="my-git-project" revision="dev" />
```

or a specific commit hash

```
<project name="my-git-project" revision="b507579809e5e5cffee5fd078e2cdae658733538" />
```

Once a project has been created, you can go to its configuration page to select a new branch of the manifest repository. When you save your changes, CodeScene will run `repo init -b <branch>` and `repo sync`, which may take some time depending on the size of your project. If you try to switch to a branch that does not contain a version of your manifest file, CodeScene will issue a warning and return you to the previous branch.

Please note that when new branches are added to your manifest repository, CodeScene will not detect them until *repo init* is run, either before an analysis or when selecting another branch.

Because of how Repo works, *Active Branch analysis* is not currently available for Repo-based projects.

With Repo, the inclusion of new Git projects does not go through the normal channels. As a result, CodeScene does not at this time automatically generate an **Architectural Component** for each Git repository. For the same reasons, and because by design the list of Git repositories in a project will evolve over time, CodeScene does not validate Architectural Components against the files present on the file system.

Duplicate project roots

Projects managed with repo tend to be large, containing many individual repositories, or *projects*, in repo's vocabulary. Projects in repo have distinct filesystem paths (either in the *name* or the *path* attribute), which means that multiple individual projects can have the same name (the last part of the path), as long as their paths are different:

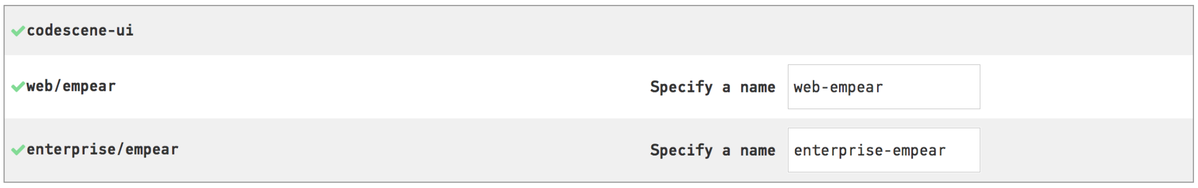
```
/path/to/a/project
/path/to/another/project
/etc/project
```

CodeScene uses project names, and not paths, to identify projects. And this means that conflicts are possible. CodeScene's repo support is designed so that adding and removing projects from the manifest file does not require any user intervention. CodeScene just follows along. In some, usually rare, cases, CodeScene has to rename projects. This can be important when using Architectural Components, Exclusion Filters or Temporal Coupling Filters that rely on a repository's project root.

To disambiguate project names in this scenario, CodeScene generates its own project names from the paths. The paths in the example above would result in the following repositories being used

```
path-to-a-project
path-to-another-project
etc-project
```

On project creation, when duplicate project roots are detected, CodeScene allows you to select your own names if you prefer.



✓codescene-ui		
✓web/empear	Specify a name	web-empear
✓enterprise/empear	Specify a name	enterprise-empear

Fig. 5.18: Renaming Repo projects to avoid name conflicts

Whether you choose your own names or use those suggested automatically, these names will be preserved. In other words, if the above paths are present on project creation, `/etc/project` will always be mapped to `etc-project`, even if it is no longer a duplicate, that is if the other repositories named `project` are removed from the manifest file.

This behavior only applies to project creation. Later, the manifest file may evolve and new name conflicts may appear at any time, each time an analysis is run. In those cases, the automatically generated name will be used, and their persistence cannot be guaranteed.

For example, if these paths are added to the manifest:

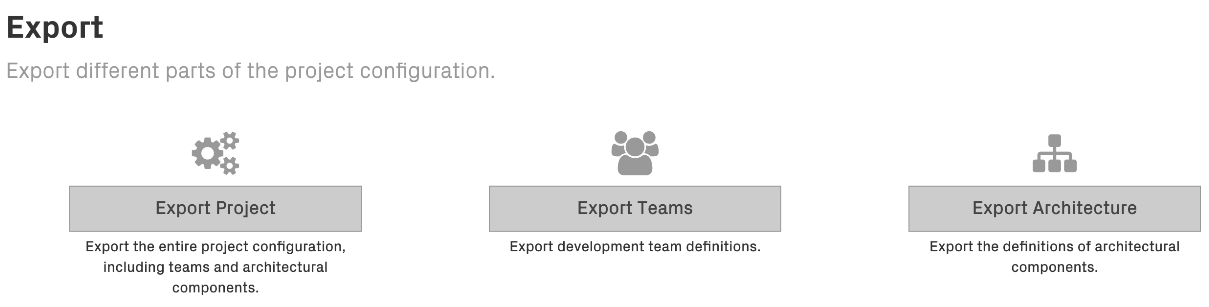
```
/a/new-project
/another/new-project
```

they will automatically become `a-new-project` and `another-new-project`. If one of them is removed, the other will revert to its original name, ie. back to `new-project`.

In some even more rare cases, there can be a conflict between the derived name of a duplicate project, like `a-new-project` and an existing, non-duplicated project that just happens to have the same name. In these cases, `a-new-project` will be renamed to `a-new-project-1` (or `a-new-project-2` etc.).

5.1.16 Exporting the project configuration

On the Export tab, the entire project configuration or parts of it can be exported to downloadable JSON and CSV files. This can be used for sharing the project configuration to another CodeScene instance, or for archiving projects before deletion.



Export

Export different parts of the project configuration.

Export Project

Export the entire project configuration, including teams and architectural components.

Export Teams

Export development team definitions.

Export Architecture

Export the definitions of architectural components.

5.2 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.

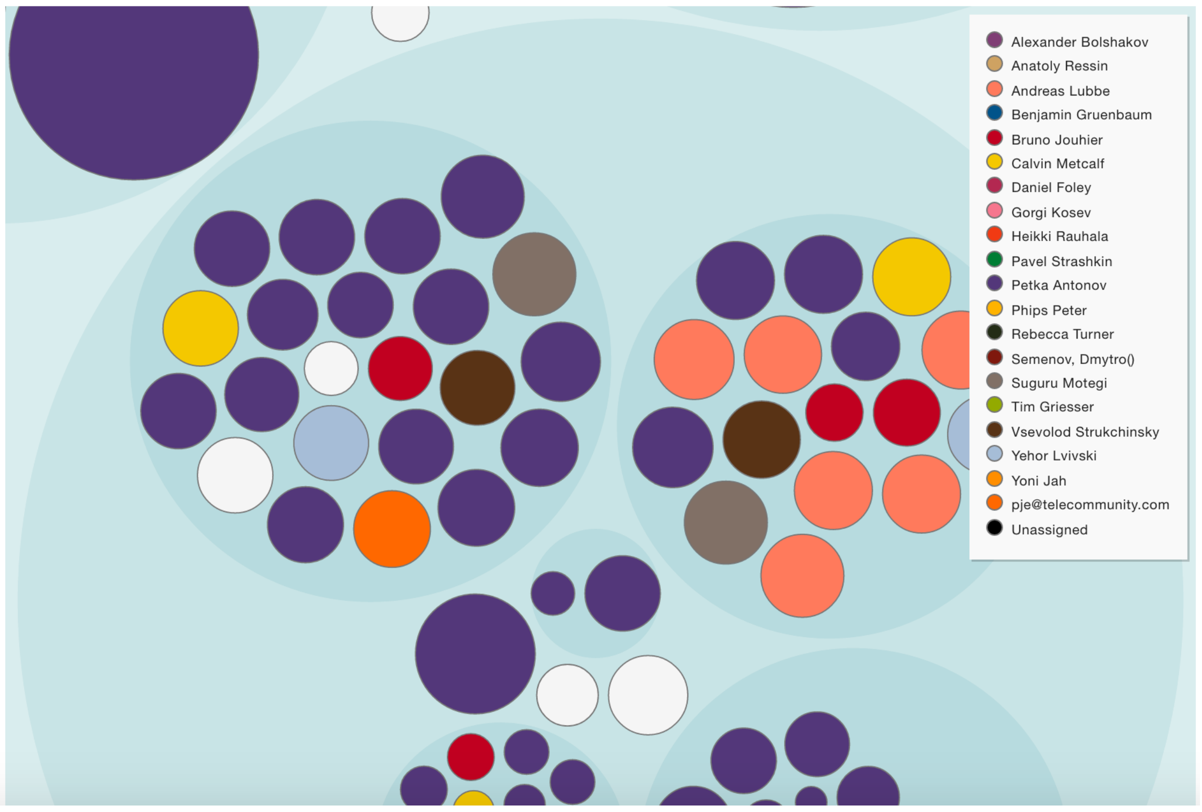


Fig. 5.19: Sample on colored knowledge maps.

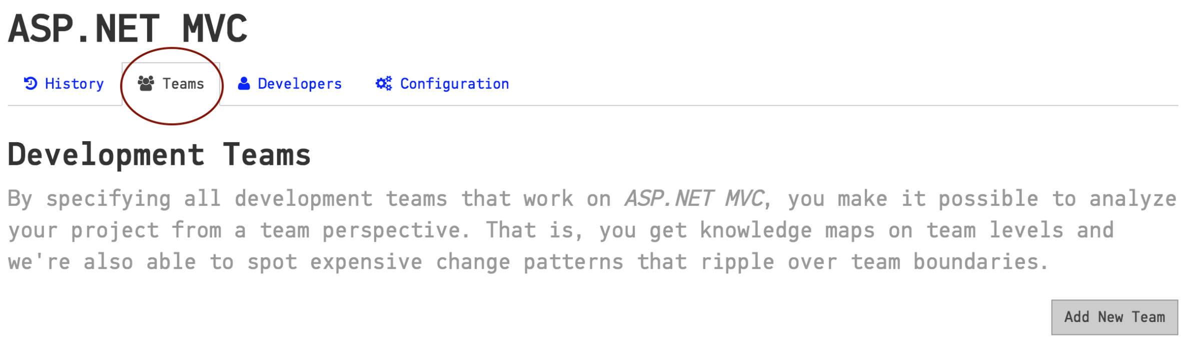
The rest of this guide will walk you through the configuration.

5.2.1 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!

5.2.2 Define Your Development Teams

Click the *Teams* tab in your project configuration to proceed to the teams configuration, as shown in Fig. 5.20.

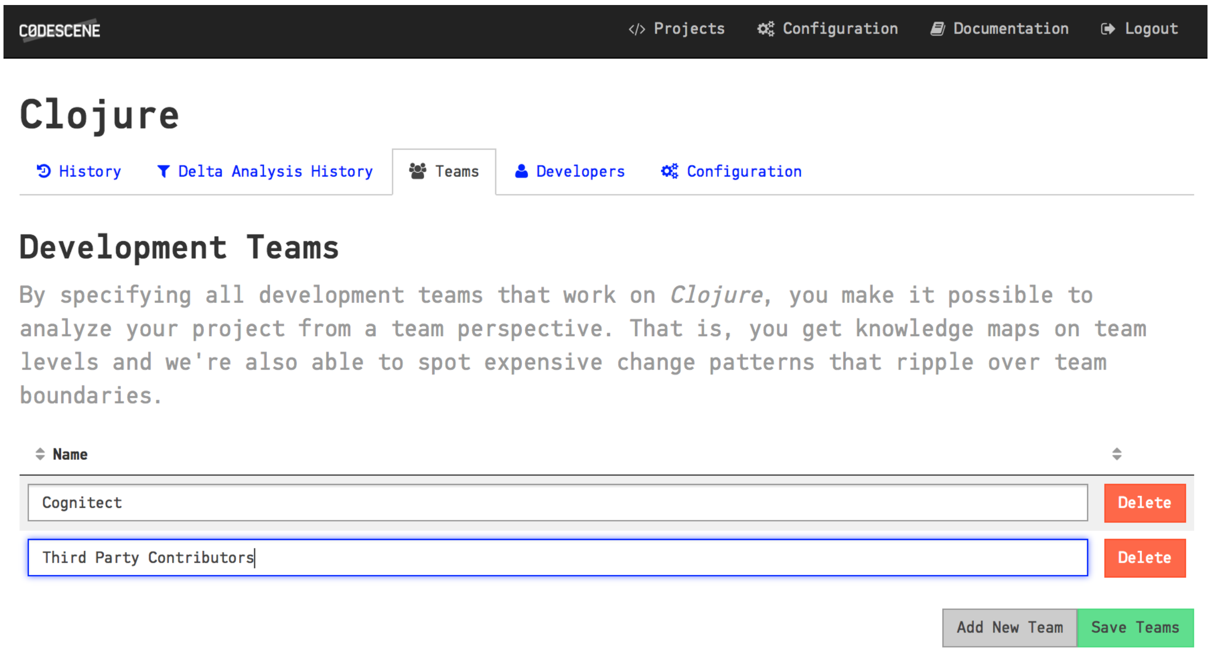

ASP.NET MVC

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. 5.20: Configure teams for a project in the Teams tab.

The only thing you have to do is to specify the name of each team in your organization. Later, when you configure developers, you'll assign them to the team names you chose here (see Fig. ??).



Clojure

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	
Cognitect	Delete
Third Party Contributors	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

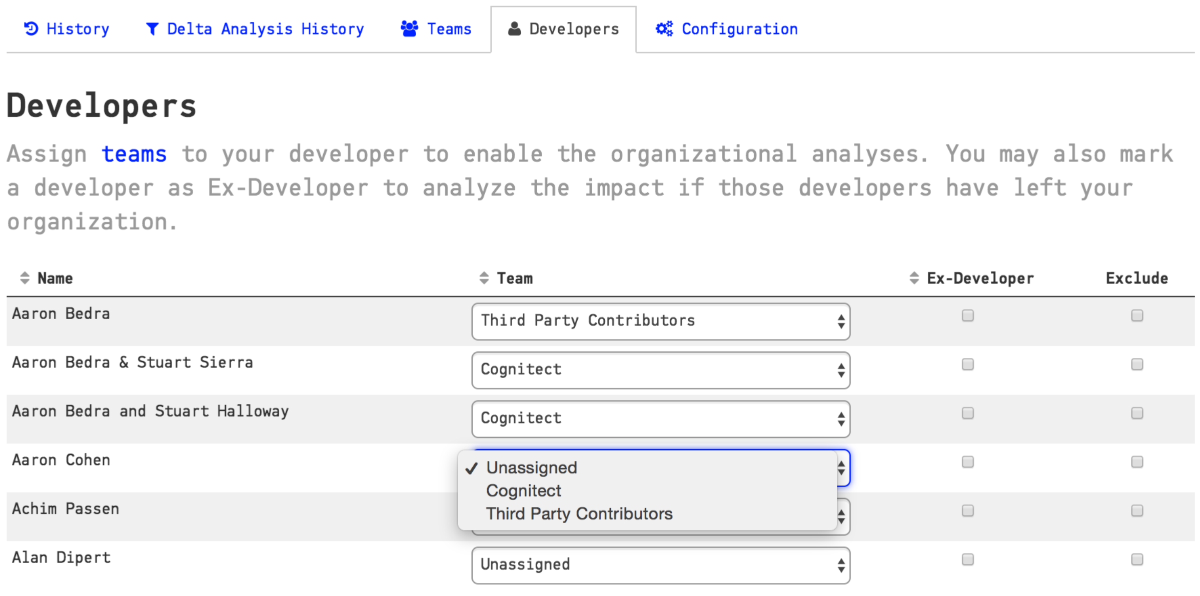
The team definition is straightforward if you analyze a codebase that's owned by a traditional organization; Just use the information from your organizational chart. However, we find it interesting to apply teams to open-source codebases as well.

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.

5.2.3 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. 5.21.



Developers

Assign teams to your developer to enable the organizational analyses. You may also mark a developer as Ex-Developer to analyze the impact if those developers have left your organization.

Name	Team	Ex-Developer	Exclude
Aaron Bedra	Third Party Contributors	☐	☐
Aaron Bedra & Stuart Sierra	Cognitect	☐	☐
Aaron Bedra and Stuart Halloway	Cognitect	☐	☐
Aaron Cohen	✓ Unassigned / Cognitect / Third Party Contributors	☐	☐
Achim Passen		☐	☐
Alan Dipert	Unassigned	☐	☐

Fig. 5.21: 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. *Active/Ex-Developer:* By default, all developers are considered active. If some of them leave your project, mark them as *Ex-Developers* and CodeScene will include them in the *Knowledge Loss Analysis*.
2. *Team:* The second column lets you assign the developer to a team. This enables CodeScene's organizational analyses such as the *Team Knowledge Distribution Analysis*.
3. *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.

5.2.4 Developers and their Aliases: Mapping Version-Control Names to People

Often, over the lifetime of a project, some developers will sign their commits with different names. This can be a source of inaccuracies for CodeScene's social analysis tools.

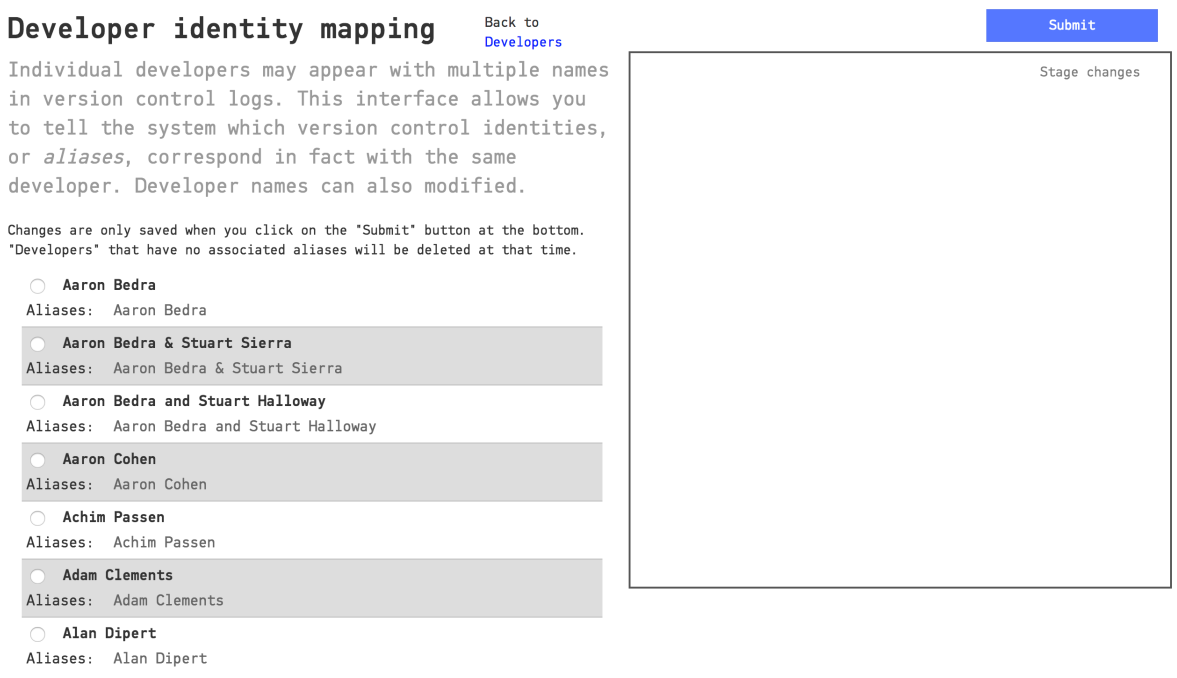
To deal with this, CodeScene provides an interface that allows you to specify the version-control names that correspond with real people. In CodeScene, when we talk about a *Developer*, we mean the real person. Team membership, author exclusion and ex-developer status belong to the developer. Each developer has at least one *Alias*, which is how they are identified in version control.

For example, a *developer* named **Jane Doe** might have several *aliases* in version control commits: **Doe, Jane**, **janedoe**, **J. Doe**, etc. This interface allows theses aliases all to refer to the same person, which provides more meaningful results in social analyses and unifies the information we have about the developer in question.

Workflow

To access the interface, click on the *Developer identity mapping interface* link near the top of the *Developers* page.

On the left, the interface displays a list of the current developers.



Developer identity mapping

Individual developers may appear with multiple names in version control logs. This interface allows you to tell the system which version control identities, or *aliases*, correspond in fact with the same developer. Developer names can also modified.

Changes are only saved when you click on the "Submit" button at the bottom.
"Developers" that have no associated aliases will be deleted at that time.

Aaron Bedra	
Aliases:	Aaron Bedra
Aaron Bedra & Stuart Sierra	
Aliases:	Aaron Bedra & Stuart Sierra
Aaron Bedra and Stuart Halloway	
Aliases:	Aaron Bedra and Stuart Halloway
Aaron Cohen	
Aliases:	Aaron Cohen
Achim Passen	
Aliases:	Achim Passen
Adam Clements	
Aliases:	Adam Clements
Alan Dipert	
Aliases:	Alan Dipert

Fig. 5.22: The Developer panel

Choose a developer you want to work on. This is the name that you want to *keep*.

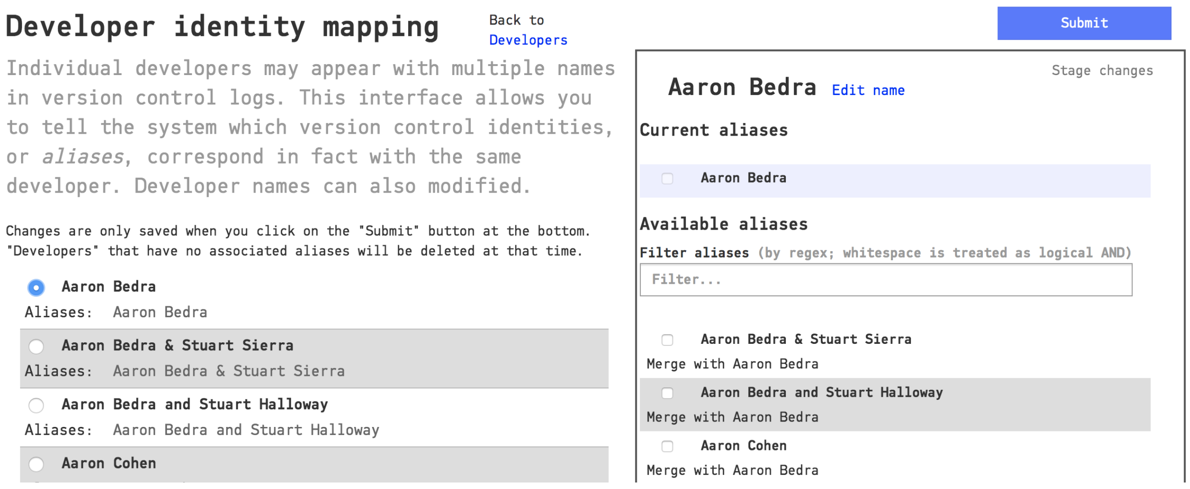
On the right, a new list will appear. Here you can add (merge) or remove (separate) aliases from the developer you selected.

In this example, we might want to merge the aliases “Aaron Bedra & Stuart Sierra” and “Aaron Bedra and Stuart Halloway” with “Aaron Bedra”. After selecting those two aliases, we would click on the *Stage changes* button. This updates the list of developers on the left. Now we can either make other modifications or click on “Submit” at the top of the window to finalize the operation.

Separating aliases from their developers

If we change our minds, we can later **separate** these aliases from the developer that we assigned them to. To do this, we select the corresponding checkbox and click on *Stage changes* again.

After clicking on *Submit*, the aliases we chose to separate will become full-fledged developers. Because these are new identities, group membership, ex-developer status, and exclusion status will be lost. Merging and unmerging an alias is *lossy*.


Developer identity mapping

Back to Developers

Individual developers may appear with multiple names in version control logs. This interface allows you to tell the system which version control identities, or *aliases*, correspond in fact with the same developer. Developer names can also modified.

Changes are only saved when you click on the "Submit" button at the bottom. "Developers" that have no associated aliases will be deleted at that time.

[✓ Aaron Bedra]
Aliases: Aaron Bedra

Aaron Bedra & Stuart Sierra
Aliases: Aaron Bedra & Stuart Sierra

Aaron Bedra and Stuart Halloway
Aliases: Aaron Bedra and Stuart Halloway

Aaron Cohen

Submit

Aaron Bedra Edit name

Stage changes

Current aliases

Aaron Bedra

Available aliases

Filter aliases (by regex; whitespace is treated as logical AND)

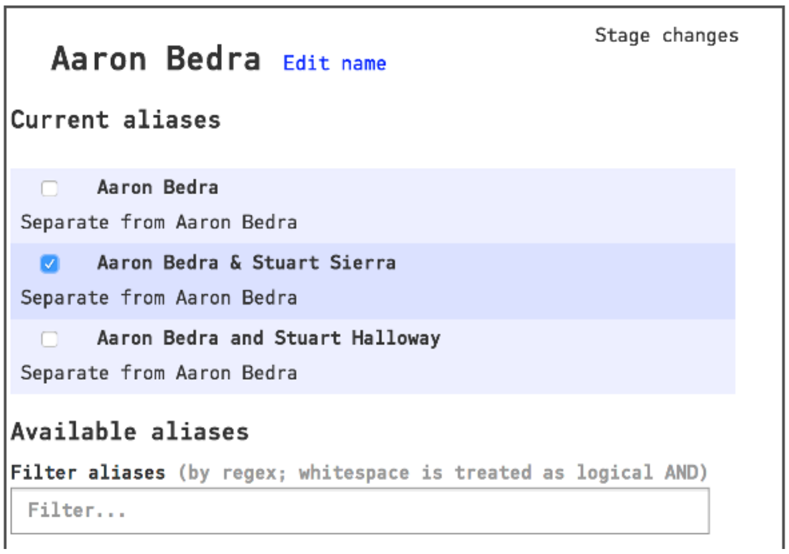
Filter...

Aaron Bedra & Stuart Sierra
Merge with Aaron Bedra

Aaron Bedra and Stuart Halloway
Merge with Aaron Bedra

Aaron Cohen
Merge with Aaron Bedra

Fig. 5.23: When developer is selected, the aliases appear


Aaron Bedra Edit name

Stage changes

Current aliases

Aaron Bedra
Separate from Aaron Bedra

[✓ Aaron Bedra & Stuart Sierra]
Separate from Aaron Bedra

Aaron Bedra and Stuart Halloway
Separate from Aaron Bedra

Available aliases

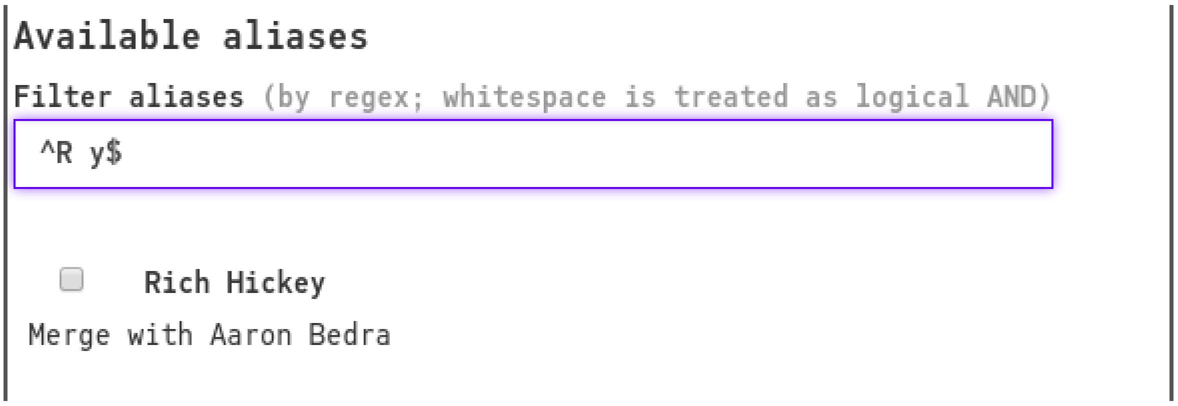
Filter aliases (by regex; whitespace is treated as logical AND)

Filter...

Fig. 5.24: Separating an alias from a developer.

Finding aliases

You can use the "Filter aliases" box to search for matching aliases. Regular expressions are allowed, with whitespace counting as a logical OR.



Available aliases

Filter aliases (by regex; whitespace is treated as logical AND)

```
^R y$
```

Rich Hickey

Merge with Aaron Bedra

Fig. 5.25: Using a regular expression to filter aliases

Renaming developers

Because the **Developer** is separate from the version control **alias**, developers can be renamed without changing how they are detected by CodeScene's analyses. To change a developer's name, click on that developer in the left column. You will notice an "Edit name" link next to their name in the box on the right.

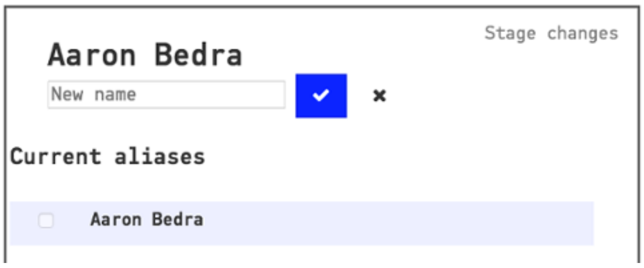


Fig. 5.26: Editing a developer's name

5.2.5 Configure for Pair Programming

In case your organization uses practices like pair or mob programming you need to tell CodeScene about it. You do that in the *Social* part of your project configuration as shown in Fig. 5.27.

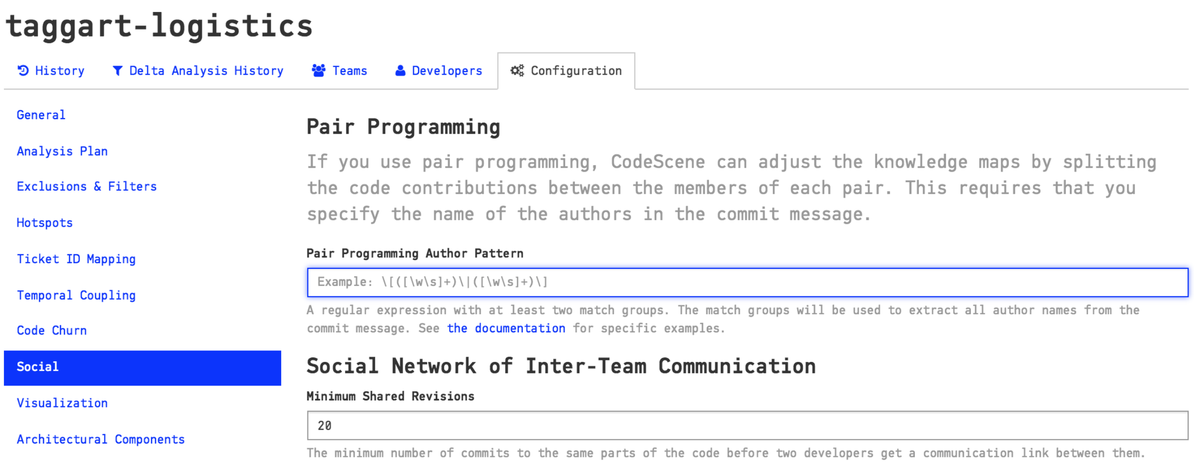
The pair and mob programming support requires that you specify the name or aliases of the authors in your commit messages. CodeScene will then extract those authors as shown in Fig. 5.28.

Using the configured pattern, CodeScene extracts the author information from your commit messages and adjusts the knowledge maps by splitting the code contributions between the members of each pair.

The configuration is based on a regular expression with the following constraints:

1. It must contain at least one match group.
2. Each match group will map to exactly one author.

Most pair programming patterns contain some kind of delimiter in the commit message. The preceding examples used square brackets for the pair programming info, *[* and *]*, and a pipe */* to separate the authors, but CodeScene supports any delimiter like *Pair: X,Y* or *(devs: X/Y)*.


taggart-logistics

Pair Programming

If you use pair programming, CodeScene can adjust the knowledge maps by splitting the code contributions between the members of each pair. This requires that you specify the name of the authors in the commit message.

Pair Programming Author Pattern

Example: \[([\w\s]+)\|([\w\s]+)\]

A regular expression with at least two match groups. The match groups will be used to extract all author names from the commit message. See the documentation for specific examples.

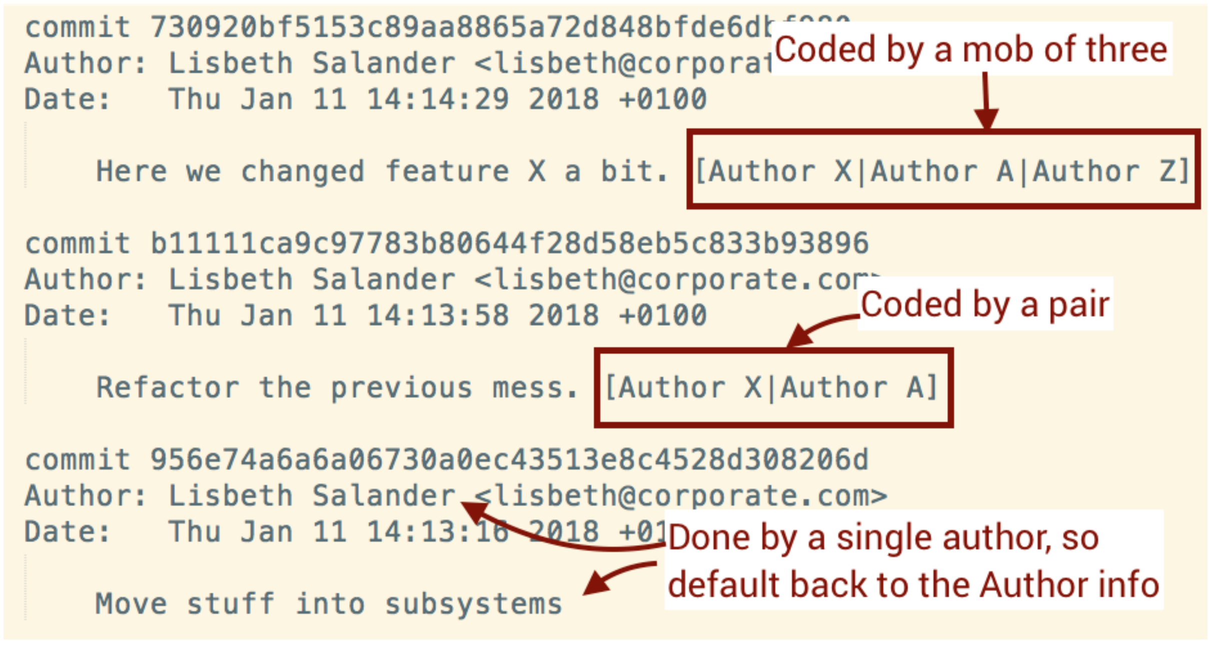
Social Network of Inter-Team Communication

Minimum Shared Revisions

20

The minimum number of commits to the same parts of the code before two developers get a communication link between them.

Fig. 5.27: Configure patterns to extract author information that reflects pair programming.


```
commit 730920bf5153c89aa8865a72d848bfde6db
Author: Lisbeth Salander <lisbeth@corporat
Date:   Thu Jan 11 14:14:29 2018 +0100

    Here we changed feature X a bit. [Author X|Author A|Author Z]

commit b11111ca9c97783b80644f28d58eb5c833b93896
Author: Lisbeth Salander <lisbeth@corporate.cor
Date:   Thu Jan 11 14:13:58 2018 +0100

    Refactor the previous mess. [Author X|Author A]

commit 956e74a6a6a06730a0ec43513e8c4528d308206d
Author: Lisbeth Salander <lisbeth@corporate.com>
Date:   Thu Jan 11 14:13:16 2018 +01

    Move stuff into subsystems
```

Fig. 5.28: Pair and mob programming annotations in the commit messages.

The most common patterns are:

- *Always a pair*: Specify a pattern such that you get two match groups. For example, to match the authors in *[Author X/Author Y]* you use a pattern like *[((ws]+)/([ws]+)]*.
- *Sometimes a pair, sometimes an individual*: CodeScene defaults to Git's *Author* information field if it cannot match the configured pattern, so this scenario will work with the previous pattern.
- *All author info is in the commit message*: In this case you need to make the second match group optional. For example, to match both the pair *[Author X/Author Y]* and the single developer *[Author X]* you specify *[([ws]+)/?([ws]+)?]*.
- *Many authors (a mob)*: To match more than two authors, we recommend that you introduce even more optional match groups. For example, to match *[Author X/Author Y/Author Z]* you specify *[([ws]+)/?([ws]+)?/?([ws]+)?]*.

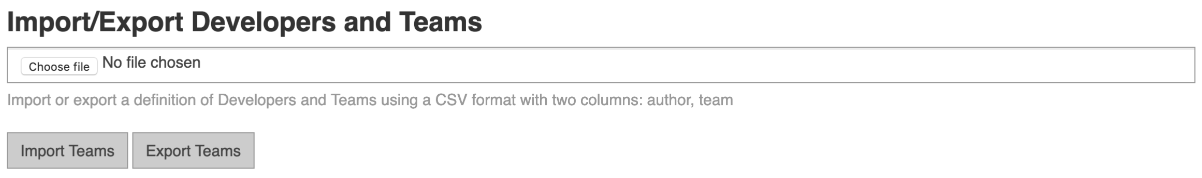
Those more elaborate patterns may be a bit tricky, but it's a one off configuration so once you have it up and running you won't see it again.

Finally, note that the preceding examples use aliases for each author instead of their full names. You can map those aliases to real author names using CodeScene's UI for developer identity aliases.

5.2.6 Import or Export a Definition of Development Teams

It may well be impractical to configure each team and developer via the UI, particularly for large organizations. That's why CodeScene supports importing the team definitions by uploading a csv-file specifying the organization. You can also share teams between projects, by exporting your team definitions and then importing them in another project.

You will find the import and export functionality in the Team configuration:



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

Fig. 5.29: Importing developer information by uploading a CSV file.

The input file specifies your organization. The file format used is a CSV with two columns: author and team.

5.3 Users and Roles

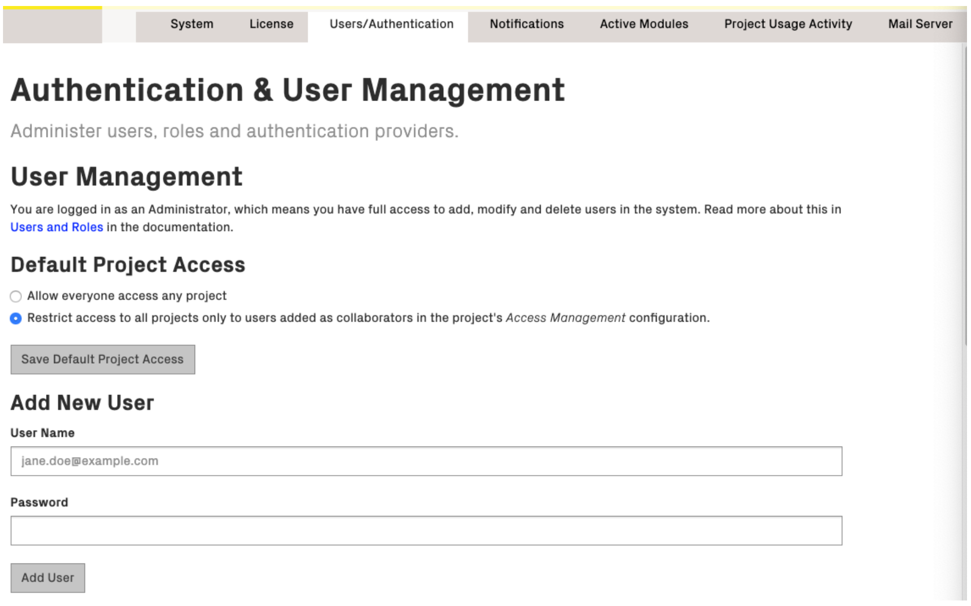
CodeScene lets you create users and grant them various levels of access depending on their roles.

5.3.1 Adding Users

When logging in with your CodeScene Username and License 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 on *Configuration* and the *Users/Authentication* tab in the top navigation bar, you can access the Users configuration page. If you are logged in as the administrator, you should see the Users configuration, as in Fig. 5.30.

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 by the users themselves.



Authentication & User Management

Administer users, roles and authentication providers.

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.

Default Project Access

( ) Allow everyone access any project

(●) Restrict access to all projects only to users added as collaborators in the project's *Access Management* configuration.

Save Default Project Access

Add New User

User Name: jane.doe@example.com

Password:

Add User

Fig. 5.30: In the global configuration you can add new users to the system.

5.3.2 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

Technical Lead
: Technical, architectural and social analyses; plan goals;

Architect
: Technical, architectural and social analyses; plan goals; project configuration

Test Leader
: Hotspot and knowledge map analyses.

Manager
: Technical quality guide and social analyses.

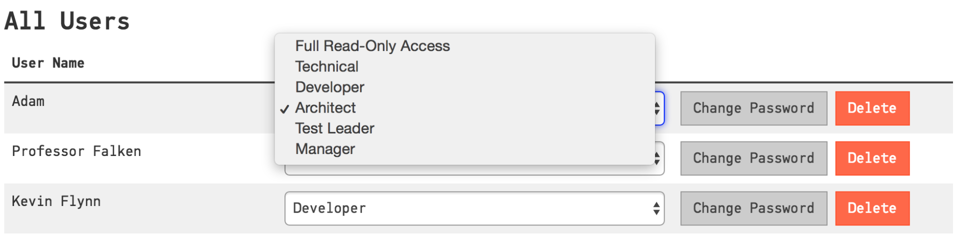
Full Read-only Access
: All analysis results, but cannot perform any actions. Typically used to display a monitor dashboard.

Bot
: This role is intended for third-party integrations like code review or continuous integration bots. This role is allowed to trigger an analysis and access the overview of the result.

In the table of existing users you can see the currently assigned roles. Click on the *Role* select box, as shown in Fig. 5.31, to change the assigned role of a user.

5.3.3 Permissions by Role

This is a more detailed description of various permissions associated with the CodeScene roles.


All Users

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

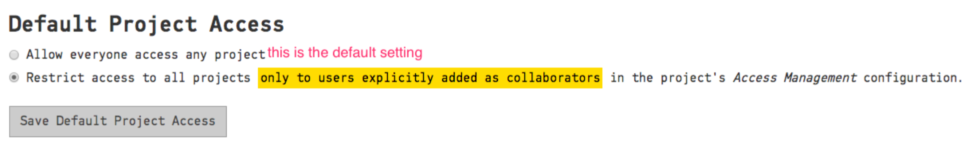
Fig. 5.31: By clicking the Role select box you can change the assigned role of a user.

Role	Permissions
Technical	• *Change own password* • *Technical analyses* - warnings, hotspots, temporal coupling, code churn trends
Developer	Same as *Technical* plus: • *Analysis process branches* (branch statistics in Project Management -> Console) • *Social analyses* - networks, knowledge map, parallel development, code churn by author, warnings, modus operandi • *Architectural analyses* - hotspots, temporal coupling
Architect	Same as *Developer* plus: • *Project configuration* including **Access Management** but not authorised to **Delete** projects • *Run a project analysis* • *Project management* - *Costs* and *Risks* in Project Management • *Analysis monitor* (Project config -> History -> Monitor) • *Off-boarding simulation*
Test Leader	• *Change own password* • *Analysis overview* • *Technical analyses* - hotspots • *Social analyses* - knowledge map
Manager	• *Change own password* • *Analysis overview* • *Analysis process branches* • *Technical analyses* - hotspots • *Social analyses* - networks, knowledge map, parallel development, code churn by author, warnings, modus operandi • *Project management* - *Costs* and *Risks* • *Analysis monitor* • *Off-boarding simulation*
Full Read-only Access	• *Analysis overview* • *Analysis process branches* • *Technical analyses* (same as *Technical*) • *Social analyses* (same as *Developer*) • *Architectural analyses* (same as *Developer*) • *Project management* (same as *Architect*) • *Analysis monitor*
Bot	

5.3.4 Project Access Management

Global Configuration

By default, all projects are visible to all CodeScene users. You can change this setting by selecting "Restrict access to all projects . . . " in the global configuration as shown in Fig. ??.



Default Project Access

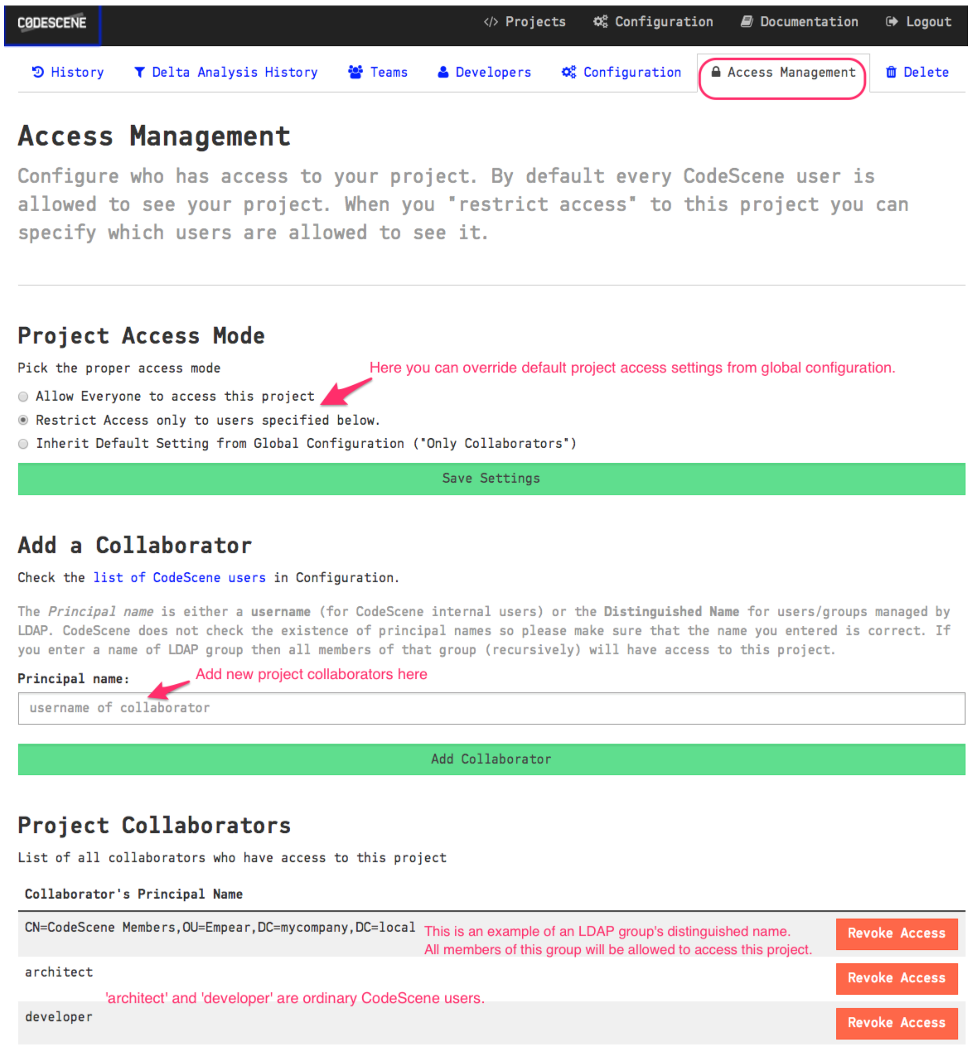
○ Allow everyone access any project this is the default setting

[✓ ● Restrict access to all projects only to users explicitly added as collaborators in the project's *Access Management* configuration.]

When access is restricted, only 'project collaborators' are allowed to access a project. Read more about project collaborators in the next section.

Project-specific Configuration

The administrator or users with the *Architect* role can configure project access management settings on a per-project basis in the project configuration tab `Access Management`:



Access Management

Configure who has access to your project. By default every CodeScene user is allowed to see your project. When you "restrict access" to this project you can specify which users are allowed to see it.

Project Access Mode

Pick the proper access mode

Here you can override default project access settings from global configuration.

○ Allow Everyone to access this project

[✓ ● Restrict Access only to users specified below.]

○ Inherit Default Setting from Global Configuration ("Only Collaborators")

Add a Collaborator

Check the list of CodeScene users in Configuration.

The *Principal name* is either a **username** (for CodeScene internal users) or the **Distinguished Name** for users/groups managed by LDAP. CodeScene does not check the existence of principal names so please make sure that the name you entered is correct. If you enter a name of LDAP group then all members of that group (recursively) will have access to this project.

Principal name: Add new project collaborators here

username of collaborator

Project Collaborators

List of all collaborators who have access to this project

Collaborator's Principal Name		
CN=CodeScene Members,OU=Empear,DC=mycompany,DC=local	This is an example of an LDAP group's distinguished name. All members of this group will be allowed to access this project.	Revoke Access
architect	'architect' and 'developer' are ordinary CodeScene users.	Revoke Access
developer		Revoke Access

Project Access Mode

There are three choices for `Project Access Mode`:

1. *Allow Everyone* - everyone is allowed to access the project regardless of the `Default Project Access` setting in the global configuration
2. *Restrict Access* - only project collaborators are allowed to access the project
3. *Inherit Default Setting* - use whatever project access mode is set in the global configuration.

Note: The administrator can always access all projects.

:: _project-collaborators: Project Collaborators ````````````````

To add a normal CodeScene user as a collaborator just enter their username and click the `Add Collaborator` button. For an LDAP user, use the distinguished name of the LDAP user or some of their LDAP groups.

When a collaborator logs in, they will only be able to see projects accessible to them.

If you use the **delta analysis** API you need to add your `Bot` user to project collaborators too.

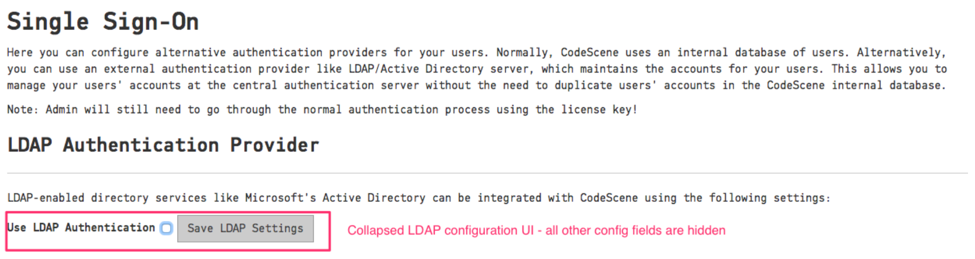
5.3.5 Single Sign-On

By default, CodeScene operates with an internal user database. Alternatively, you can configure another authentication provider, such as LDAP/Active Directory or OAuth2/OpenID Connect, to perform identity verification for your users, thus avoiding the duplication of your users' accounts in CodeScene. Users can then log in using the same credentials that they use for other services within your system. When using OAuth2/OpenID Connect, the users are redirected to the configured provider to authorize CodeScene at first login.

LDAP Authentication Provider

A generic LDAP server or Active Directory can be used for user authentication.

LDAP authentication is turned off by default and the configuration fields are hidden as shown in Fig. **??**.


Single Sign-On

Here you can configure alternative authentication providers for your users. Normally, CodeScene uses an internal database of users. Alternatively, you can use an external authentication provider like LDAP/Active Directory server, which maintains the accounts for your users. This allows you to manage your users' accounts at the central authentication server without the need to duplicate users' accounts in the CodeScene internal database.

Note: Admin will still need to go through the normal authentication process using the license key!

LDAP Authentication Provider

LDAP-enabled directory services like Microsoft's Active Directory can be integrated with CodeScene using the following settings:

Use LDAP Authentication ☐

Save LDAP Settings
Collapsed LDAP configuration UI - all other config fields are hidden

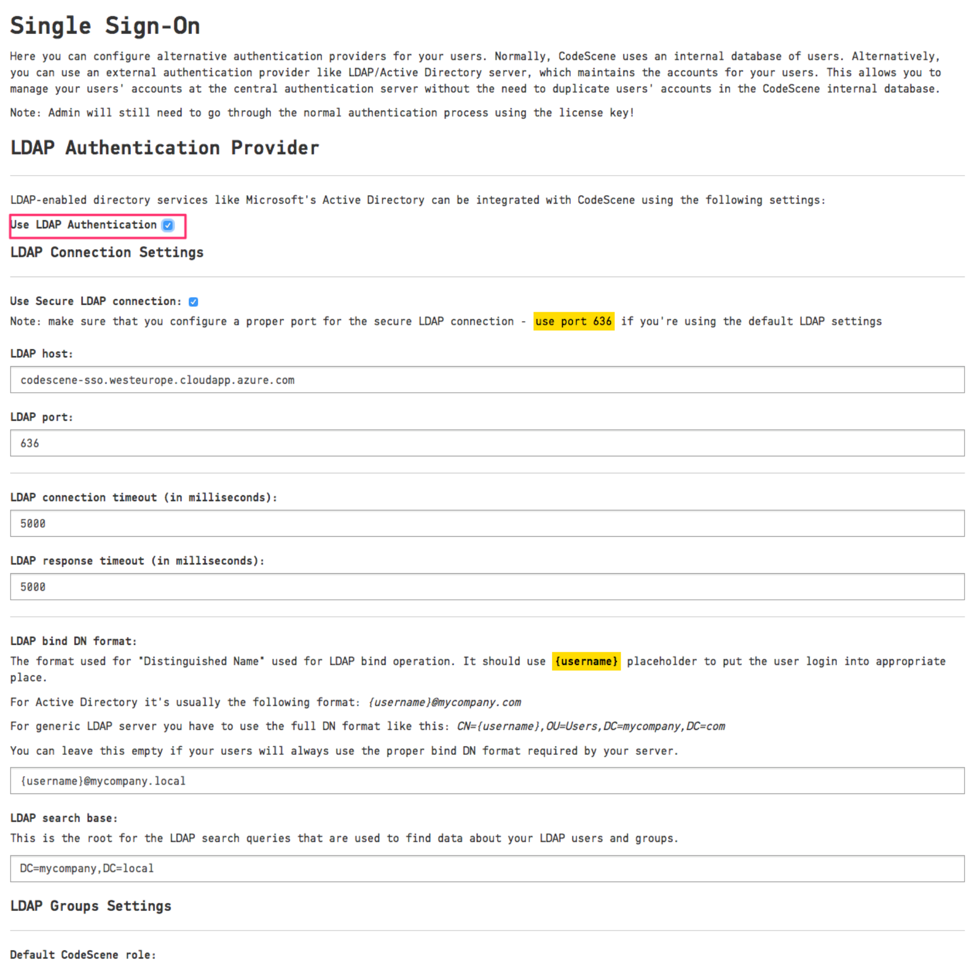
Activate LDAP Authentication by clicking on the "Use LDAP Authentication" checkbox and fill in the details as shown in Fig. **??**.

You will need to configure the "LDAP host" address and the "LDAP search base" settings. CodeScene provides default values for the remaining settings, e.g. port and connection timeouts.

The "LDAP search base" is used as a root for LDAP queries searching for data about users and their groups. Make sure to specify a proper base for the search to not miss any relevant user data. See Components of an LDAP Search: for more details.

The "LDAP Bind DN format" is used to create a proper full login name accepted by your LDAP server. It's usually a full "Distinguished Name", although Active Directory supports various formats like the "User Principal Name" (e.g. username@mycompany.com) or *sAMAccountName*. You will use *{username}* placeholder to configure the username expansion - see the examples on the Configuration page. You can leave this field empty if your users always enter the full login name manually.

We also encourage you to use the "Secure LDAP" connection by checking the "Use Secure LDAP connection" checkbox. In this case, you will need to change the LDAP port too; secure LDAP connections often use port 636.



Single Sign-On

Here you can configure alternative authentication providers for your users. Normally, CodeScene uses an internal database of users. Alternatively, you can use an external authentication provider like LDAP/Active Directory server, which maintains the accounts for your users. This allows you to manage your users' accounts at the central authentication server without the need to duplicate users' accounts in the CodeScene internal database.

Note: Admin will still need to go through the normal authentication process using the license key!

LDAP Authentication Provider

LDAP-enabled directory services like Microsoft's Active Directory can be integrated with CodeScene using the following settings:

Use LDAP Authentication ✓

LDAP Connection Settings

Use Secure LDAP connection: ✓

Note: make sure that you configure a proper port for the secure LDAP connection - use port 636 if you're using the default LDAP settings

LDAP host:

codescene-sso.westeurope.cloudapp.azure.com

LDAP port:

636

LDAP connection timeout (in milliseconds):

5000

LDAP response timeout (in milliseconds):

5000

LDAP bind DN format:

The format used for "Distinguished Name" used for LDAP bind operation. It should use **{username}** placeholder to put the user login into appropriate place.

For Active Directory it's usually the following format: *{username}@mycompany.com*

For generic LDAP server you have to use the full DN format like this: *CN={username},OU=Users,DC=mycompany,DC=com*

You can leave this empty if your users will always use the proper bind DN format required by your server.

{username}@mycompany.local

LDAP search base:

This is the root for the LDAP search queries that are used to find data about your LDAP users and groups.

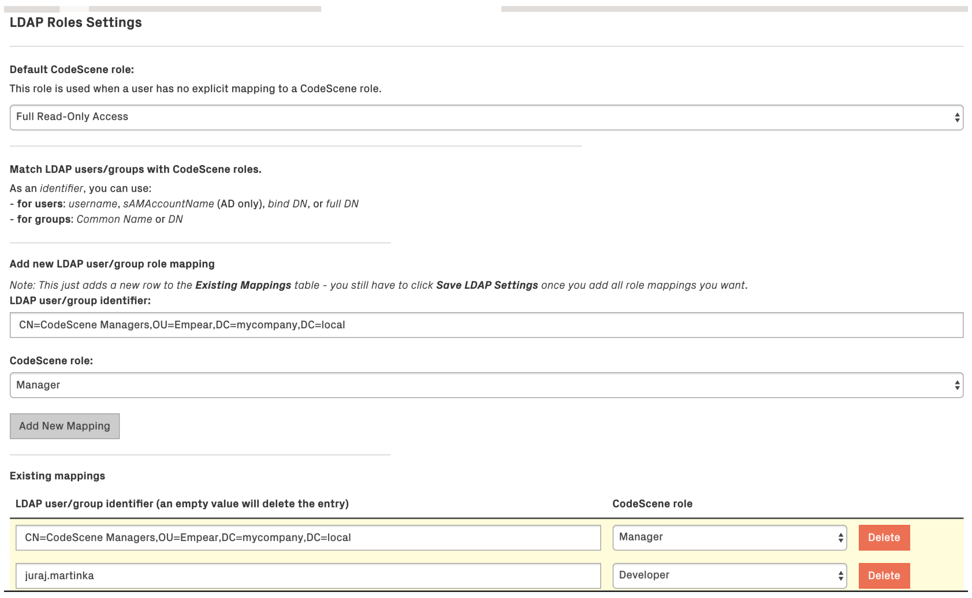
DC=mycompany,DC=local

LDAP Groups Settings

Default CodeScene role:

LDAP Roles Settings

Like normal CodeScene users, users authenticated with the LDAP authentication provider also need to have a "role" assigned to them. This is done with the "LDAP Roles Settings" as shown in Fig. **??**.


LDAP Roles Settings

Default CodeScene role:
This role is used when a user has no explicit mapping to a CodeScene role.
Full Read-Only Access

Match LDAP users/groups with CodeScene roles.
As an *identifier*, you can use:
- **for users**: *username*, *sAMAccountName* (AD only), *bind DN*, or *full DN*
- **for groups**: *Common Name* or *DN*

Add new LDAP user/group role mapping
Note: This just adds a new row to the ***Existing Mappings*** *table - you still have to click* ***Save LDAP Settings*** *once you add all role mappings you want.*
LDAP user/group identifier:
CN=CodeScene Managers,OU=Empear,DC=mycompany,DC=local

CodeScene role:
Manager

Add New Mapping

Existing mappings

LDAP user/group identifier (an empty value will delete the entry)	CodeScene role	
CN=CodeScene Managers,OU=Empear,DC=mycompany,DC=local	Manager	Delete
juraj.martinka	Developer	Delete

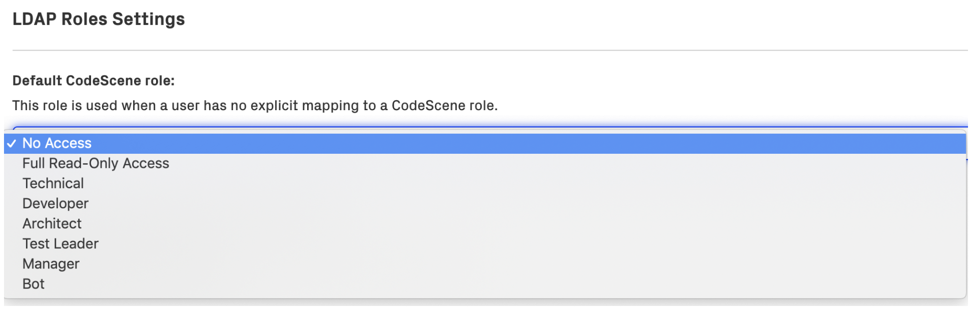
When user data is fetched from an LDAP server, the user's "identifiers" and his LDAP groups are matched to CodeScene roles based on the "LDAP Roles Settings" configuration:

- To identify a user, you can use his username, sAMAccountName (Active Directory only), bind DN, or full DN.
- To identify a group, you can use full DN or Common Name - note that using Common Name isn't recommended and is provided only for backward compatibility reasons (the Common Name attribute isn't guaranteed to be unique and it isn't possible to use it in Project Access Management).

In the example configuration, you can see that the user *juraj.martinka* has the role *Developer* and all members of the LDAP group *CN=CodeScene Managers,OU=Empear,DC=mycompany,DC=local* are *Manager*-s.

Nested groups are supported; that is if the LDAP user is a member of the group "Managers" which is a member of the group "CodeScene Managers" then that LDAP user will have the CodeScene's *Manager* role too.

If no matching CodeScene role is found for the LDAP user, the value of "Default CodeScene role" is used. By default, this is set to *Full Read-Only Access*, but it can be changed to a more restrictive role or even a special *No Access* role which will deny access to all LDAP users who aren't mapped explicitly. You can see this in Fig. **??**.

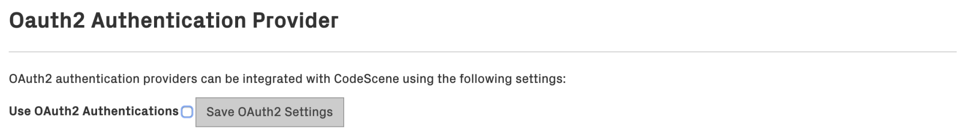

LDAP Roles Settings

Default CodeScene role:
This role is used when a user has no explicit mapping to a CodeScene role.

OAuth2 Authentication Provider

A generic OAuth2/Open ID Connect server can be used for user authentication. CodeScene supports the Authorization Code Grant flow, and uses the token received through the authorization process to access user and team info from configurable URLs at the OAuth2 provider.

OAuth2 authentication is turned off by default and the configuration fields are hidden as shown in Fig. **??**.

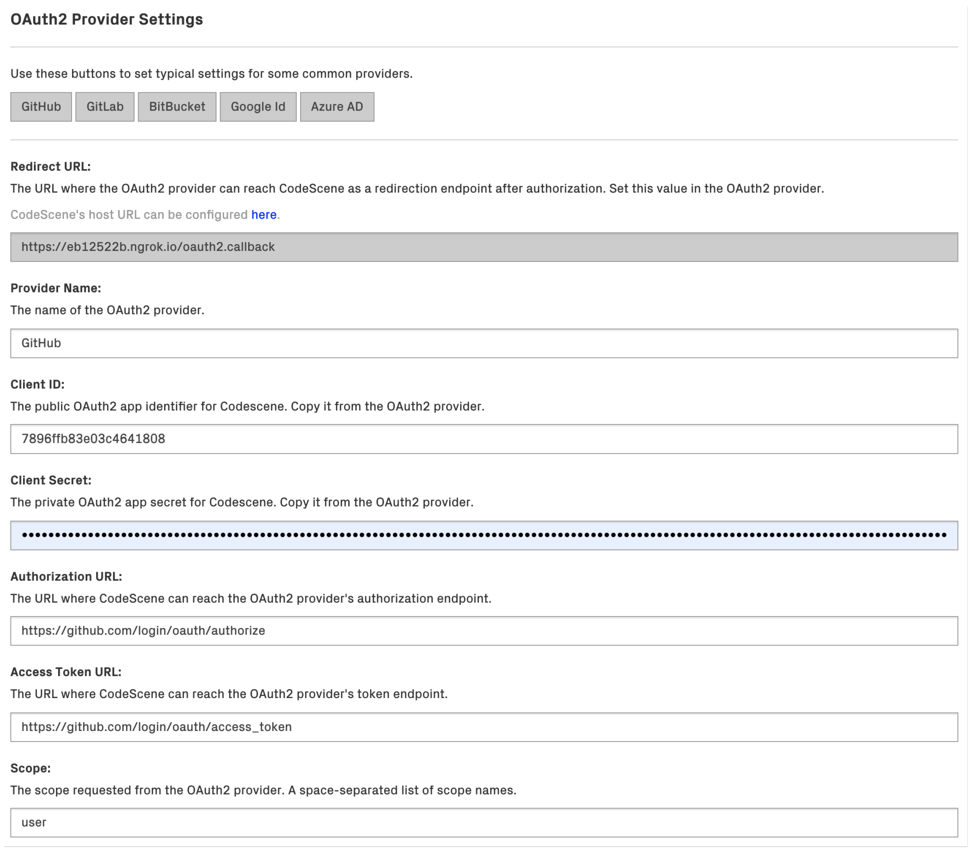


Oauth2 Authentication Provider

OAuth2 authentication providers can be integrated with CodeScene using the following settings:

Use OAuth2 Authentications ☐ Save OAuth2 Settings

Activate OAuth2 Authentication by clicking on the "Use OAuth2 Authentication" checkbox and fill in the details as shown in Fig. **??**.



OAuth2 Provider Settings

Use these buttons to set typical settings for some common providers.

GitHub | GitLab | BitBucket | Google Id | Azure AD

Redirect URL:
The URL where the OAuth2 provider can reach CodeScene as a redirection endpoint after authorization. Set this value in the OAuth2 provider.
CodeScene's host URL can be configured here.

https://eb12522b.ngrok.io/oauth2.callback

Provider Name:
The name of the OAuth2 provider.

GitHub

Client ID:
The public OAuth2 app identifier for Codescene. Copy it from the OAuth2 provider.

7896ffb83e03c4641808

Client Secret:
The private OAuth2 app secret for Codescene. Copy it from the OAuth2 provider.

••

Authorization URL:
The URL where CodeScene can reach the OAuth2 provider's authorization endpoint.

https://github.com/login/oauth/authorize

Access Token URL:
The URL where CodeScene can reach the OAuth2 provider's token endpoint.

https://github.com/login/oauth/access_token

Scope:
The scope requested from the OAuth2 provider. A space-separated list of scope names.

user

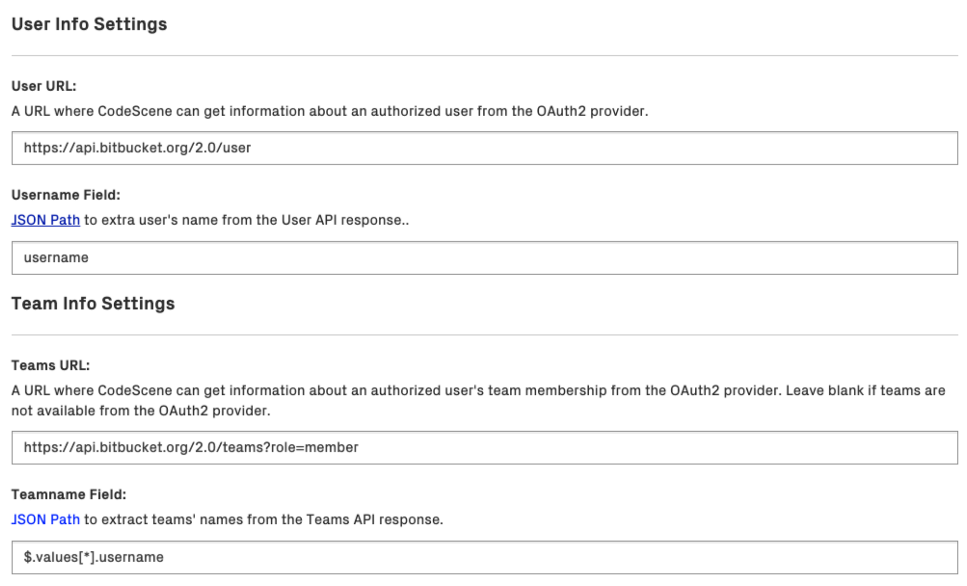
You will need to register CodeScene as an application/consumer at the OAuth2 provider. The provider will then create a Client ID and Secret that you use when configuring the provider in CodeScene. The other settings are specific for each provider, and the buttons at the top can be used to set these to typical settings for some common providers. Note that the scope requested must be such that it gives access to the specified user/team URLs and fields.

The *User URL* setting supports URLs like the OpenID Connect UserInfo endpoint, i.e. an URL that, when given an access token, returns claims for the logged in user.

Similarly, the *Teams URL* supports URLs that, when given an access token, return a list of the teams that the logged in user is a member of.

Note about the *User Name* and *Team Name* fields:

- The *User Name* field is a JSON Path string that selects an attribute in the user API response that represents a user's identifier.


User Info Settings

User URL:
A URL where CodeScene can get information about an authorized user from the OAuth2 provider.
https://api.bitbucket.org/2.0/user

Username Field:
JSON Path to extra user's name from the User API response..
username

Team Info Settings

Teams URL:
A URL where CodeScene can get information about an authorized user's team membership from the OAuth2 provider. Leave blank if teams are not available from the OAuth2 provider.
https://api.bitbucket.org/2.0/teams?role=member

Teamname Field:
JSON Path to extract teams' names from the Teams API response.
$.values[*].username

- The *Team Name* field is a JSON Path string that selects names of *all* teams in the teams API response. Some providers return teams as a top-level JSON array while others (like BitBucket and Azure AD) return them wrapped in another top-level JSON object. Check the typical settings for providers.

E.g. Bitbucket Teams API may return following API response:

```
{
  "pagelen" : 10,
  "values" : [ {
    "username" : "empear",
    "display_name" : "empear",
    "uuid" : "{f19d05a0-693c-4327-bd80-58eff5a64d04}",
    "links" : {
      "hooks" : {
        "href" : "https://api.bitbucket.org/2.0/teams/%7Bf19d05a0-693c-4327-bd80-58eff5a64d04
↪%7D/hooks"
      },
      ...
    },
    "created_on" : "2018-04-16T13:38:09.387625+00:00",
    "type" : "team",
    "properties" : { },
    "has_2fa_enabled" : null
  } ],
  "page" : 1,
  "size" : 1
}
```

To extract the team name, which is stored as the *username* field in the *values* array, we use the following JSON Path syntax: *$.values[*].username*. The *** symbol selects all the elements (teams) in the *values* array if multiple teams are returned in the response.

In summary, the steps to follow for configuring OAuth2 authentication are:

1. In CodeScene, make sure the *CodeScene Host URL* is set to a URL where CodeScene can be reached from the OAuth2 Provider.
2. Note the *Redirect URL* in CodeScene's *OAuth2 Provider Settings*.
3. At the OAuth2 provider, register an application/consumer and set the *Callback/Redirect URL* to CodeScene's *Redirect URL*.

4. Note the *Client ID* and *Client Secret* values.
5. In CodeScene, set the *Client ID* and *Client Secret* values.
6. Set the other *OAuth2 Provider Settings* to values specific for the OAuth2 provider.

OAuth2 Roles Settings

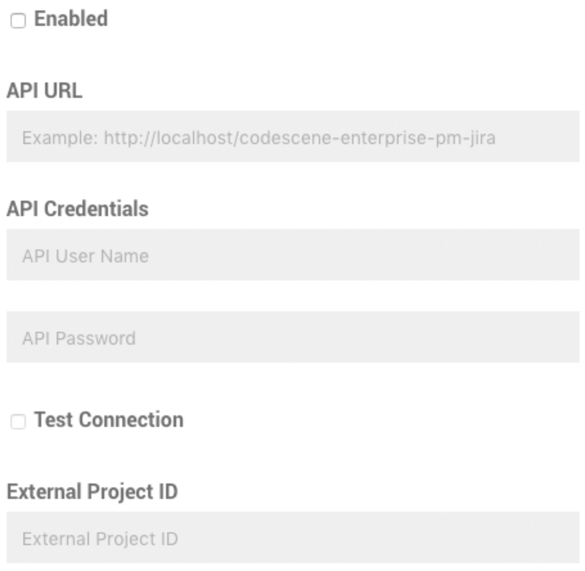
OAuth2 role settings and matching works exactly as described in *LDAP Roles Settings* (page 172) with the difference that the matching in this case is done using the username and team names acquired from the *User/Team URLs* as defined in the Oauth2 Provider Settings.

5.4 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.

5.4.1 Repository Configuration

By default, PM integration is disabled (see Fig. 5.32). 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. 5.32: Check 'Enabled' to enable the project management integration.

Enabling the integration lets you edit the remaining fields (see Fig. 5.33):

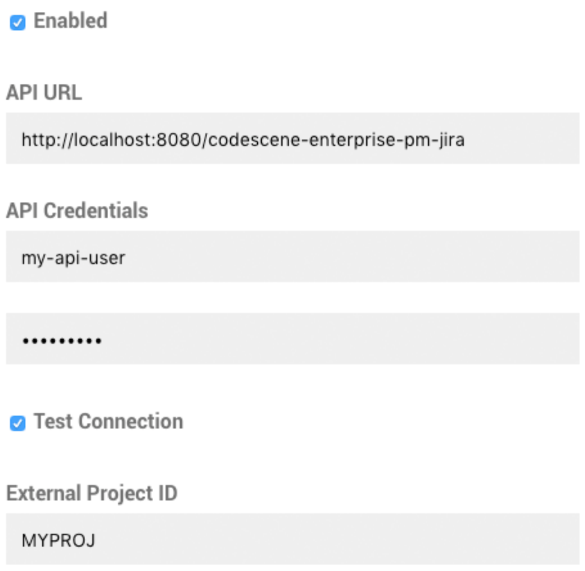
API URL
: The base URL of the PM integration service. If you have deployed the JIRA integration in Tomcat, the URL will likely be *http://localhost:8080/codescene-enterprise-pm-jira*.

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*.



☑ Enabled

API URL

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

API Credentials

my-api-user

•••••••••

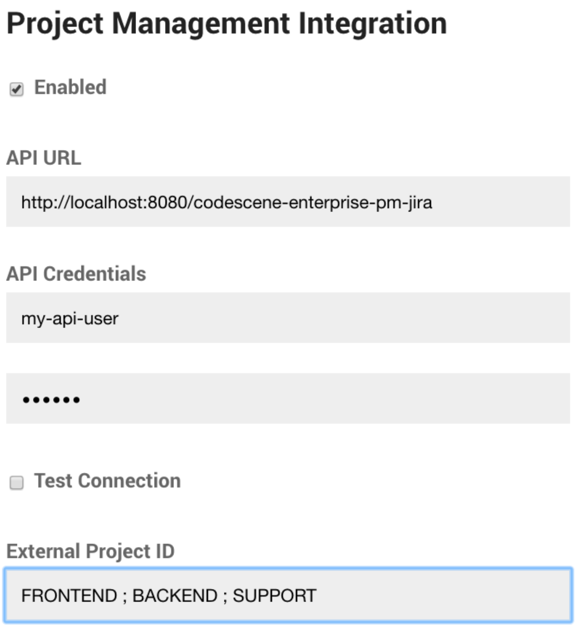
☑ Test Connection

External Project ID

MYPROJ

Fig. 5.33: A configuration sample for project management integration.

You can add multiple JIRA projects here by separating them with a semicolon, *;* as shown in Fig. 5.34



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. 5.34: A configuration sample for integration with multiple projects.

5.4.2 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*. *Tune the House-Keeping Options for Analysis Results* (page 149) explains Ticket ID Mapping in greater detail. Fig. 5.35 illustrates how both mappings extracts IDs with the same format.

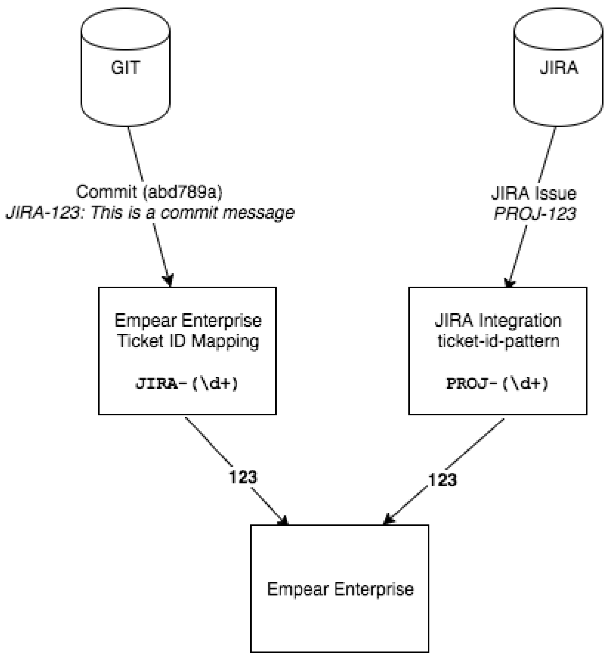


Fig. 5.35: Ticket IDs are extracted from the VCS logs using Ticket ID Mapping, and Project Management Item IDs are mapped from JIRA issue keys using a configured pattern in the JIRA integration service.

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. 5.36 shows an example on such a configuration.

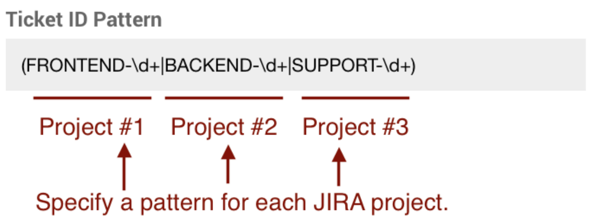

Ticket ID Pattern
(FRONTEND-\d+|BACKEND-\d+|SUPPORT-\d+)
Project #1
Project #2
Project #3
Specify a pattern for each JIRA project.

Fig. 5.36: Ticket ID specification that matches items from multiple JIRA projects.

5.5 Manage Projects

A large organization might have hundreds of CodeScene projects. This section explains how you get an overview of those projects, see which one's are the most active, and where most active authors reside. You use this information to identify inactive projects that can be removed safely in order to keep license costs down. Let's start by explaining how CodeScene's license model works.

5.5.1 What's an Active Author?

CodeScene's license is based on the number of active contributors. An active contributor is anyone who has committed code over the past three months to the codebases you want to analyse. This time period is a sliding window that always starts at the date of the most recent commit in your repositories. CodeScene applies the following additional rules:

- *Each author is only counted once.* That is, if you analyze multiple codebases, the same persons only count once no matter how many projects they contribute to.
- *Historic contributors are free.* People who haven't committed code within the last three months are included for free and don't add to the license fee.

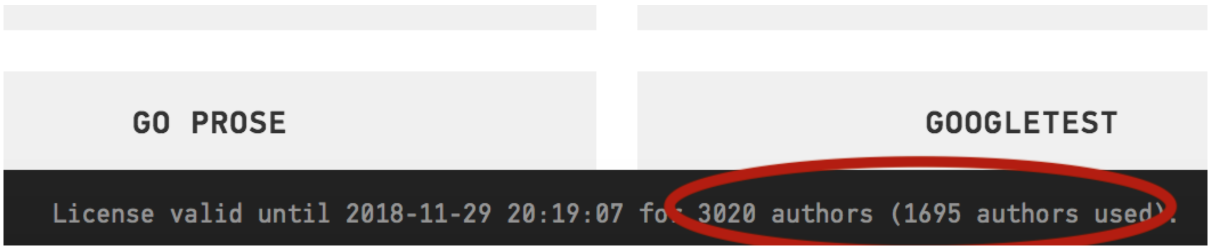
You can get a rough estimate on the number of active authors in a Git repository through the following command:

```
git shortlog -sn --after=2018-07-01
```

Note that you want to replace the *–after* date with a date that's three months back from where you are now.

5.5.2 How do I monitor Project Activity and Active Authors?

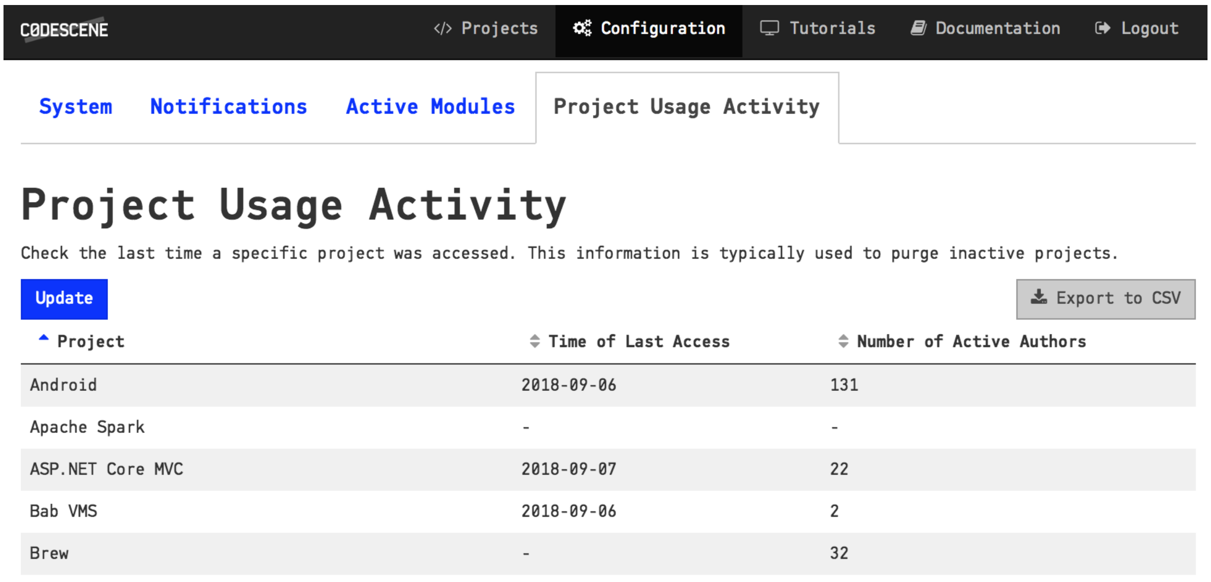
The aggregated number of active authors is shown in the footer on CodeScene's start page, as shown in Fig. 5.37.



License valid until 2018-11-29 20:19:07 for 3020 authors (1695 authors used).

Fig. 5.37: The number of active authors is shown in the footer on the front page.

In addition, CodeScene comes with a project activity view. If you login as an admin, you can inspect the project usage activity as shown in Fig. 5.38.



Project Usage Activity

Check the last time a specific project was accessed. This information is typically used to purge inactive projects.

Project	Time of Last Access	Number of Active Authors
Android	2018-09-06	131
Apache Spark	-	-
ASP.NET Core MVC	2018-09-07	22
Bab VMS	2018-09-06	2
Brew	-	32

Fig. 5.38: A summary of the project usage activity in CodeScene.

This view lets you identify inactive projects that haven't been accessed in a long time (2-3 months), and safely delete them to keep the active author count down and maybe save some licensing fees in the process; Although we're admittedly happy the more users you have, we also want the licensing to be fair. So we recommend that you inspect the project usage regularly.

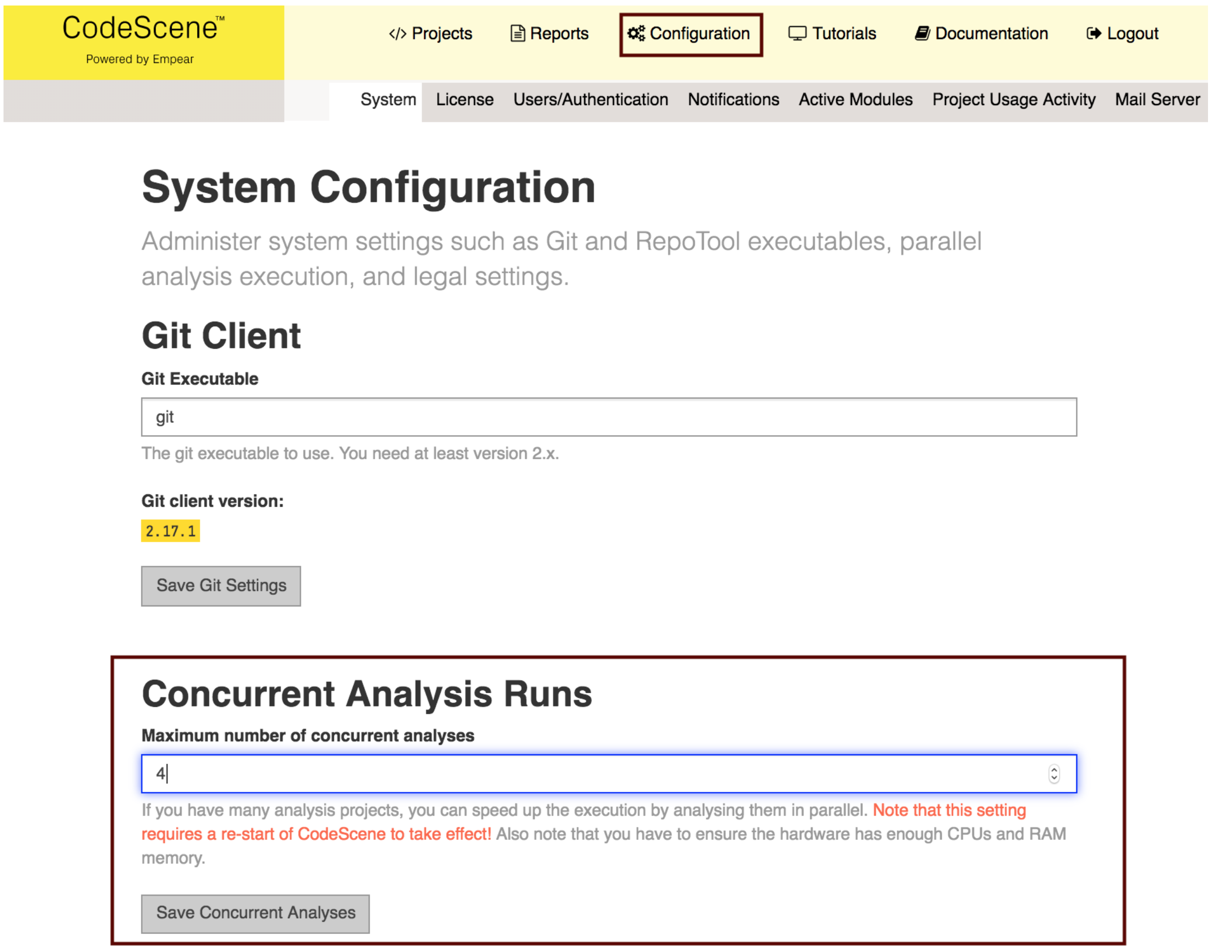
5.6 Performance: Enable Concurrent Analyses

By default, the analyses in CodeScene are executed serially. That is, an analysis starts once the previous one has completed. The rationale for the default behavior is to let CodeScene work well on limited hardware; the analyses might consume plenty of memory and CPU.

However, if you have enough hardware capabilities available, then you can run more analyses in parallel and optimize the overall analysis time.

5.6.1 Enable Concurrent Analyses

As an admin, go to the top level Configuration tab as shown in Fig. 4.113.



System Configuration

Administer system settings such as Git and RepoTool executables, parallel analysis execution, and legal settings.

Git Client

Git Executable

git

The git executable to use. You need at least version 2.x.

Git client version:

2.17.1

Save Git Settings

Concurrent Analysis Runs

Maximum number of concurrent analyses

4

If you have many analysis projects, you can speed up the execution by analysing them in parallel. Note that this setting requires a re-start of CodeScene to take effect! Also note that you have to ensure the hardware has enough CPUs and RAM memory.

Save Concurrent Analyses

Fig. 5.39: Configure the maximum number of concurrent analyses.

Note that a change setting requires a restart of the CodeScene service in this case (yes, we're sorry!).

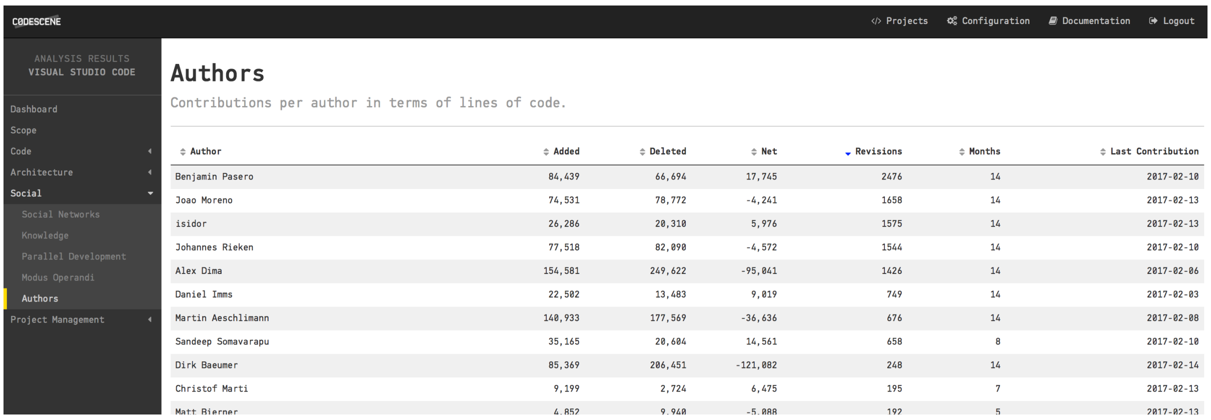
You also have to make sure you have enough system resources to execute incoming CI/CD delta analysis requests that will be processed in parallel to the analyses as well.

5.7 Legal Restrictions

Some analysis information from CodeScene may be considered sensitive from a legal perspective. This is a topic that varies between different jurisdictions and/or company policies. Thus, CodeScene provides configuration options that let you disable such information.

5.7.1 Disable the Author Statistics

CodeScene provides an aggregated view of all author contributions. This information is intended as descriptive data that lets you find long-term contributors as shown in Fig. 4.113.



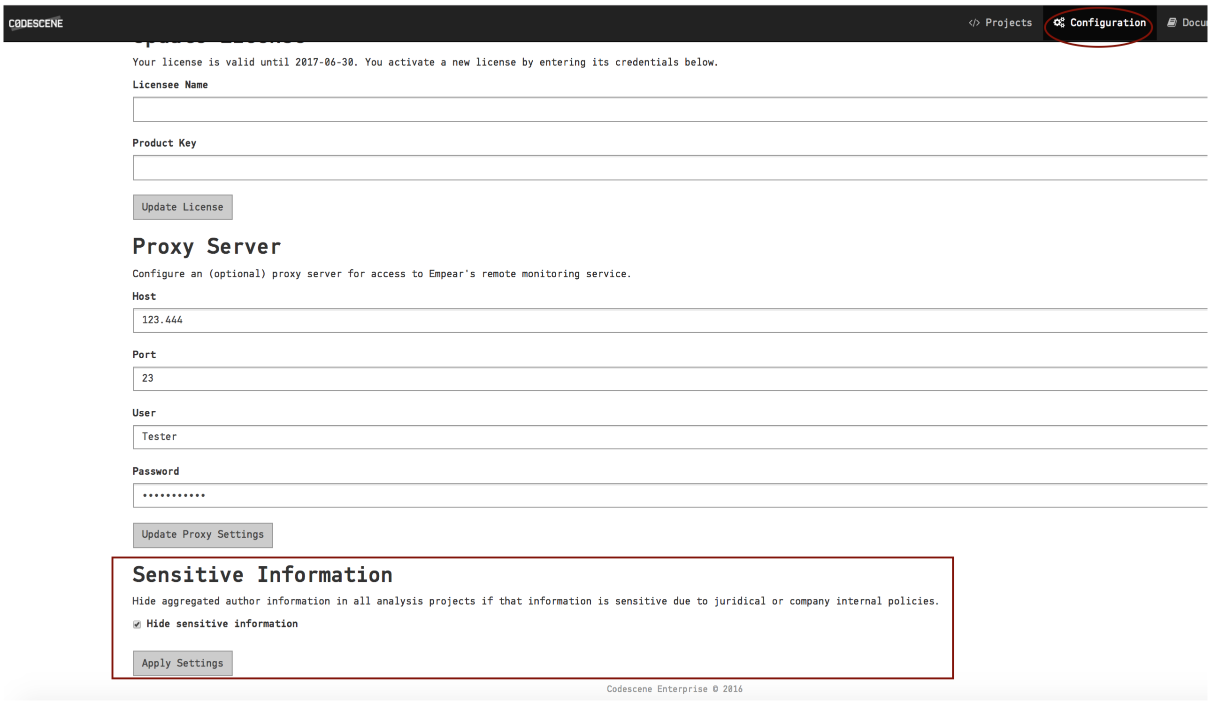
Authors

Contributions per author in terms of lines of code.

Author	Added	Deleted	Net	Revisions	Months	Last Contribution
Benjamin Pasero	84,439	66,694	17,745	2476	14	2017-02-10
Joao Moreno	74,531	78,772	-4,241	1658	14	2017-02-13
isidor	26,286	20,310	5,976	1575	14	2017-02-13
Johannes Rieken	77,518	82,090	-4,572	1544	14	2017-02-10
Alex Dima	154,581	249,622	-95,041	1426	14	2017-02-06
Daniel Imms	22,502	13,483	9,019	749	14	2017-02-03
Martin Aeschlimann	140,933	177,569	-36,636	676	14	2017-02-08
Sandeep Somavarapu	35,165	20,604	14,561	658	8	2017-02-10
Dirk Baeumer	85,369	206,451	-121,082	248	14	2017-02-14
Christof Marti	9,199	2,724	6,475	195	7	2017-02-13
Matt Bierner	4,852	9,940	-5,088	192	5	2017-02-13

Fig. 5.40: The detailed author statistics show the aggregated contributions.

You disable this analysis by logging in as an administrator, click the Configuration tab in the top bar, and check the box as shown in Fig. 5.41.



Your license is valid until 2017-06-30. You activate a new license by entering its credentials below.

Licensee Name

Product Key

Update License

Proxy Server

Configure an (optional) proxy server for access to Empear's remote monitoring service.

Host

123.444

Port

23

User

Tester

Password

•••••••••••

Update Proxy Settings

Sensitive Information

Hide aggregated author information in all analysis projects if that information is sensitive due to juridical or company internal policies.

[✓] Hide sensitive information

Apply Settings

Codescene Enterprise © 2016

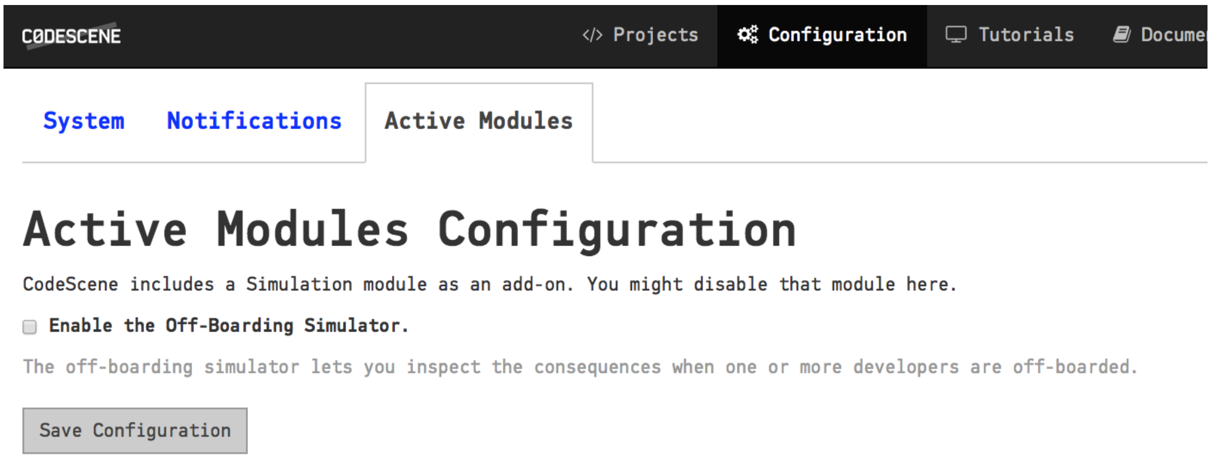
Fig. 5.41: The configuration lets a CodeScene administrator disable sensitive information.

5.7.2 Disable the Off-Boarding Simulation

The off-boarding simulation lets you simulate the impact in case individual developers leave the organization. You disable the off-boarding simulation as shown in Fig. 4.142.

5.7.3 A Warning on Performance Evaluations

The detailed author statistics are useful in order to find the people that carry the history of your codebase and product in their head. Their stories often complement the analysis results and help you put your



Active Modules Configuration

CodeScene includes a Simulation module as an add-on. You might disable that module here.

☐ Enable the Off-Boarding Simulator.

The off-boarding simulator lets you inspect the consequences when one or more developers are off-boarded.

Fig. 5.42: The off-boarding simulation can be disabled in the global configuration.

findings into context.

We strongly recommend against using this data for performance evaluations. That isn't the purpose of these analyses. The reason we advice against this is part ethical and part practical. In particular, once someone starts to evaluate contributors people will adapt by optimizing for what's being measured. For example, if I'm evaluated by how many commits I do I'll increase the number of commits. My commits will no longer carry any meaning, but my statistics "improve". In addition, using this data for performance evaluation is likely to destroy the team dynamics. Again, if I'm measured by how many commits or lines of code I produce I'm less likely to invest time in supporting my peers and we end up with local optimizations that hurt the overall productivity.