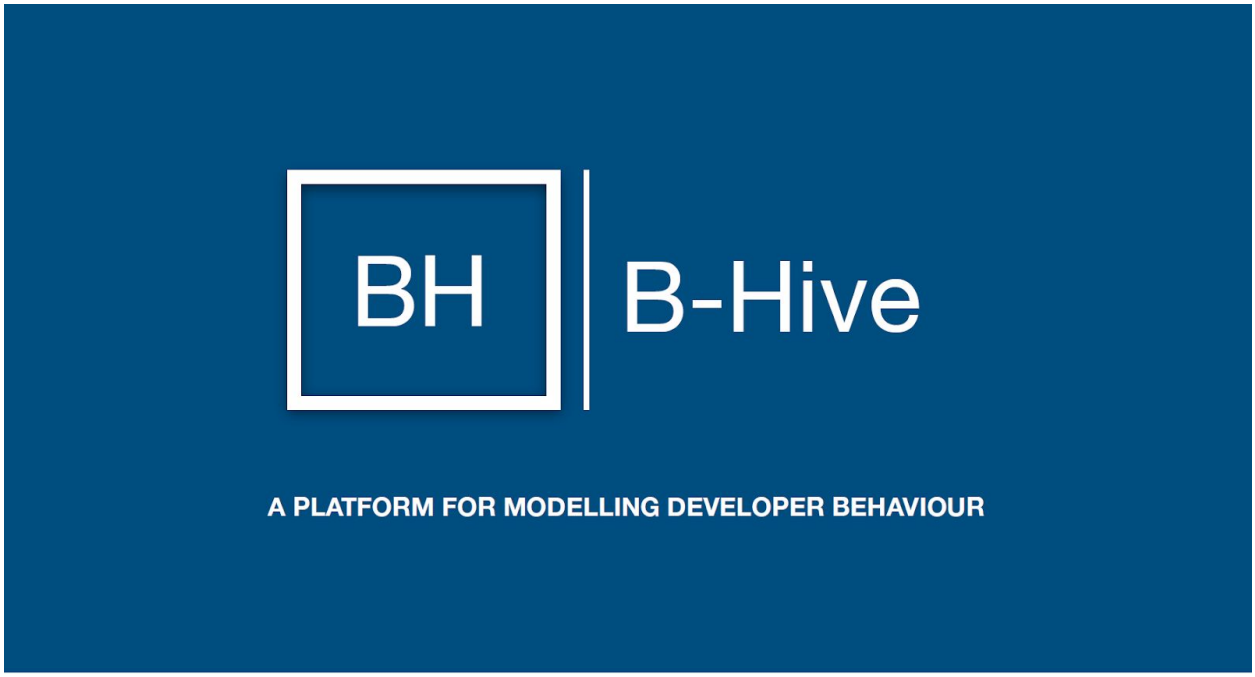
DEV FACTORY

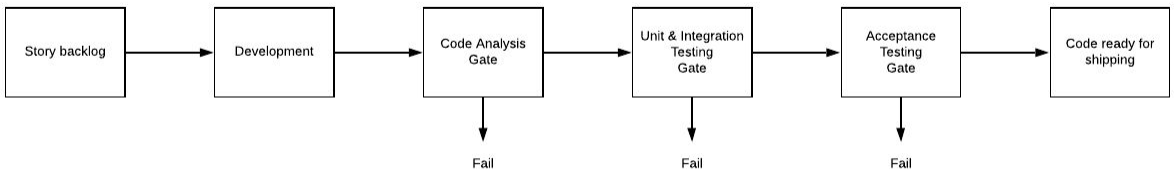


DevFactory is a member of the Trilogy group.
The contents of this paper are based on research, testing and experience with over 4000 developers and codebases from 120 technology companies totalling over 600M lines of code.

The productivity and code quality struggle

With all the advances in agile processes and automated tooling over the past 20 years, one would think the quality of our codebases would not be a topic of discussion and that the productivity of development teams would continue to increase with each additional automation or process improvement. However, organisations continue to struggle to improve quality and productivity despite advanced code analysis tools, a wide range of well known design patterns and anti-patterns and a leadership-level understanding of technical debt and its consequences.

The problem is that organisations have created a production anti-pattern. Similar to automated acceptance and unit tests, code analysis tools such as Sonar, CAST and Checkmarx, have been applied using a traditional quality process, automatically preventing code changes that fail to meet a defined quality bar from propagating down the CI/CD pipeline. When an organisation sets about improving quality, the bar is raised and additional code changes fail to propagate downstream. Quality improvements cost productivity. This gating approach identifies and rejects defective code but does nothing to reduce the number of defects coming down the line from the source, the developer. In a factory it's like rejecting finished items at the end of the line without identifying which stage during production created the defect.



Quality gates are important because they prevent escaping defects. However, our tooling needs to do more than identify defects. It must identify the source of the defect so that targeted upstream improvements can be made.

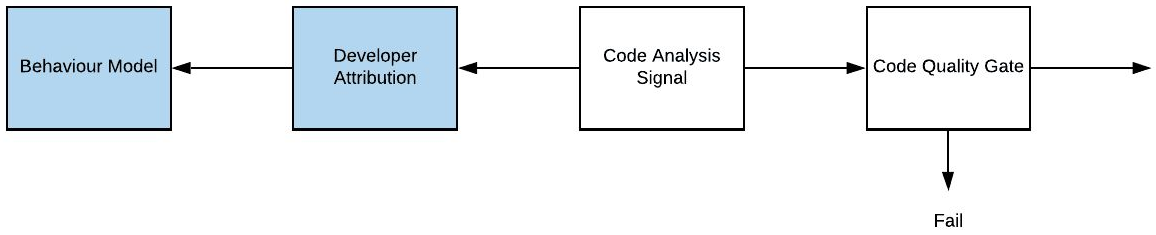
Improving code quality

At DevFactory we believe, and our experience has shown, that we gain greater productivity improvements by focusing primarily on quality.

Improving quality by addressing the defects reduces the defects in the codebase but does not improve the rate at which they are produced. A more enduring improvement can be made by identifying and improving developer behaviour, behaviour that is demonstrated, not by the existence or absence of a defect, but by a repeated pattern over a period of time.

By identifying the developers and behaviours that are creating the greatest quality issues it is possible to apply a targeted improvement which gives the best return in overall codebase quality.

This approach can use the same code analysis tools as a CI/CD pipeline. Signals about the codebase are fed back upstream, attributed to each developer that created the code and fed into a behaviour model. The output of this behaviour model can be used by developers themselves, by team managers and by organisation leadership to improve the quality of code changes.



Behaviour modelling

Trust

You'll notice that attribution of code issues to a developer is the first step and presents a significant challenge, that of trust. Developers and their managers will naturally discredit and resist acting on information, especially negative, if they don't trust it. There are two factors required for a trusted signal:

- The issue identified must be important. If the code analysis is not identifying the most significant problems in the codebase it's very easy to ignore.
- There must be a very low level of false-positives. It only takes one inaccurate result to undermine the whole collection of results.

Creating a trusted signal requires that we are selective with our code analysis tools and the results that we use from them.

Attribution

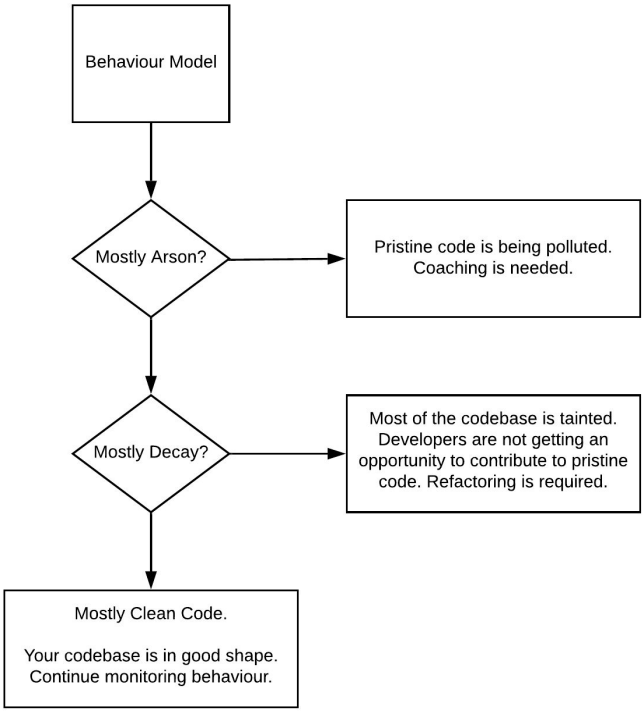
Attribution tools like git-blame assign blame for all code issues found in a file to the developer who made the latest commit. It only works when the codebase is pristine to begin with. However, most codebases already have code issues before changes are made. Our tools need to look beyond the latest commit to the history of commits, when issues were introduced or worsened and who was responsible.

It's important to differentiate new issues from existing issues. In B-Hive we have 3 types of attribution.

Clean Code	When new code is added or changes made to pristine code, we expect the pristine state to be maintained.
Decay	When code is added or changes made to tainted parts of the code, it propagates the issue and causes further decay.
Arson	Introducing issues to pristine code is value destruction. Strong language is deliberately used.

Taking Action

The Clean Code/Decay/Arson model guides the type of action we take.



Improving Low Performers

We don't have technology today that can identify and grade good code. Our tools are capable of only detecting the degradation of the code. This means that our focus for raising the average quality of a codebase must be on improving the lowest performers. At DevFactory we recommend a strategy that improves the lowest 25% by 25% each quarter and have seen dramatic improvements over time.

Data Driven Coaching

We've found, and customers agree, that less than 20% of developers will self-coach themselves to improve. For the remainder, the best way to achieve improvement for individual developers is through manager coaching. Focusing on the bottom percentile allows the manager to get the best improvement in average codebase quality for management time spent.

A weekly manager-led 15 minutes coaching session is most effective when the following data is available:

- Weekly and daily metrics for each developer
- 90 day history of metrics
- Positive/negative trend indications

Organisation level initiatives

In some cases specific code issues may be widespread and the coaching may need to be an initiative at the organization level. An organisation level view of quality metrics and trends is required across all teams. This allows a technical leader responsible for many teams to understand where improvements are needed and to initiate, for example, a training plan.

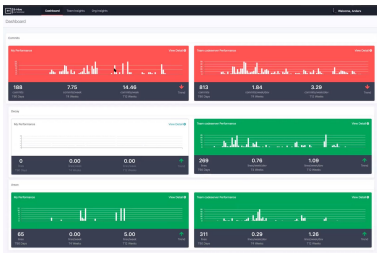
B-Hive

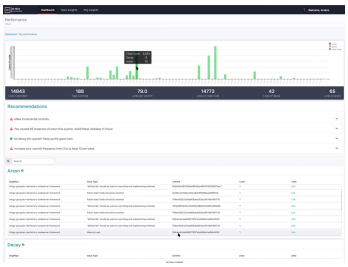
DevFactory's B-Hive is a Developer-first SaaS platform that integrates with code analysis tools and source control systems to model developer behaviour. Views are provided at Developer, Manager and Technical Leadership level and the platform can be used to build your own apps.

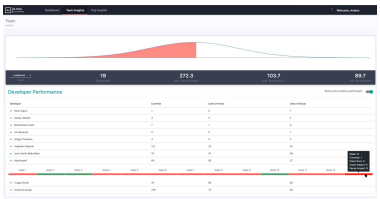
Developers can view their own performance, compare it with their team's and use the self-service coaching system to improve their code quality.

For the first time, managers have very concrete data to coach each individual along with a rolling 13 week view of progress to ensure that the behavior change is actually taking place.

At the organisation level you get data backed comparisons of issues and behaviours across teams which enable you to make macro decisions around coaching and development processes.







Frequently Asked Questions

Is B-Hive a code analysis tool?

No. B-Hive is a behaviour modelling tool. B-Hive uses signals from code analysis tools like Sonar, CAST, Checkmarx, Infer and DevFactory's CodeGraph as an input to create the behaviour model. We are adding support for new tools all the time to get new signals.

Do you also handle signals from internal processes or other non-source systems?

No. B-Hive creates a behavior model only from signals from the code (and commits).

What do I need to get started?

The minimum that you need to provide is access to your source control with the entire version history. GitHub is supported today with other git-based SCMs to come soon. Other non-git SCMs will follow.

Is B-Hive a SaaS Product?

Yes, B-Hive is a SaaS only product that is hosted on AWS