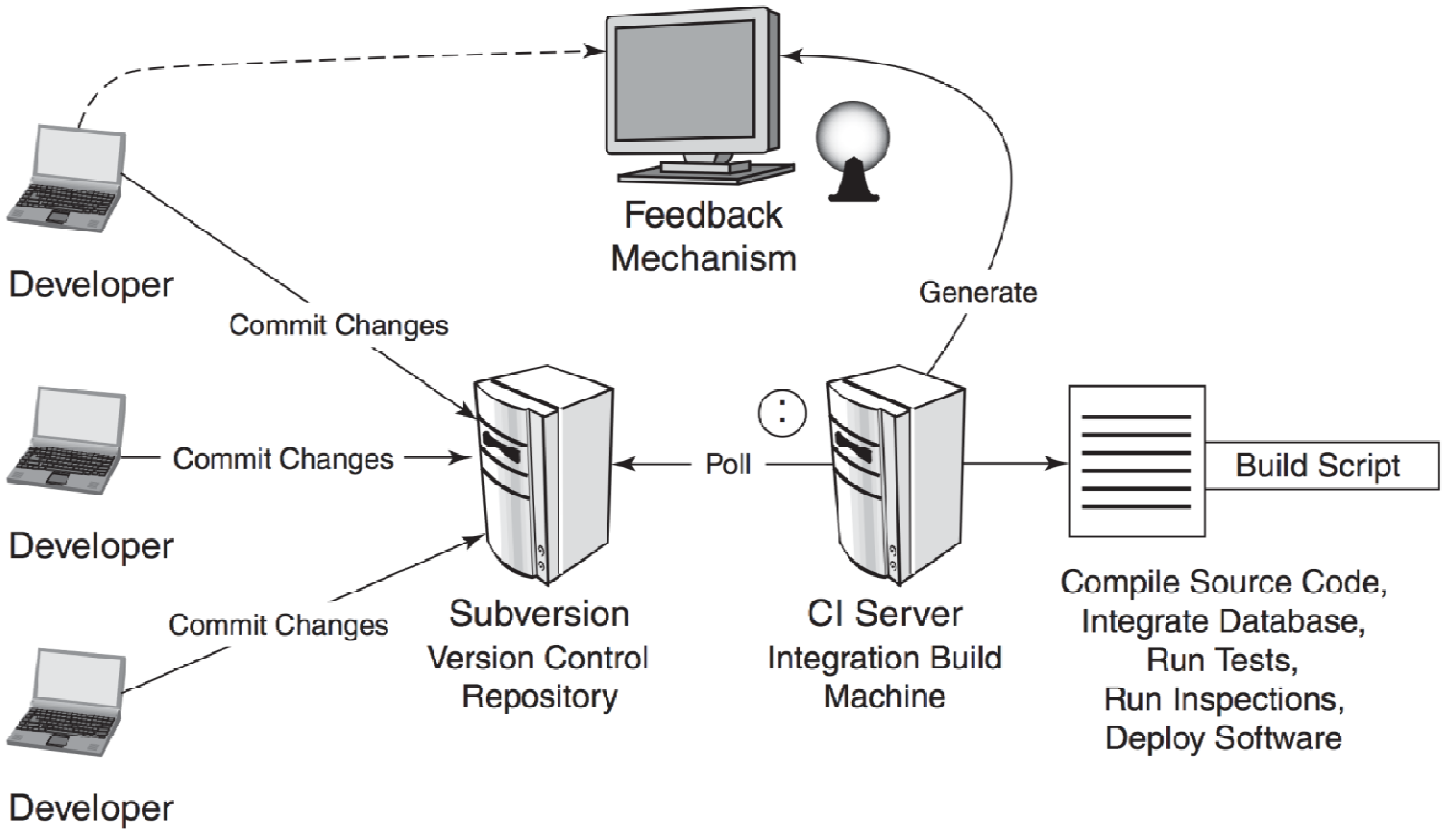
# 大规模软件开发持续集成的罪与罚

Nokia Networks

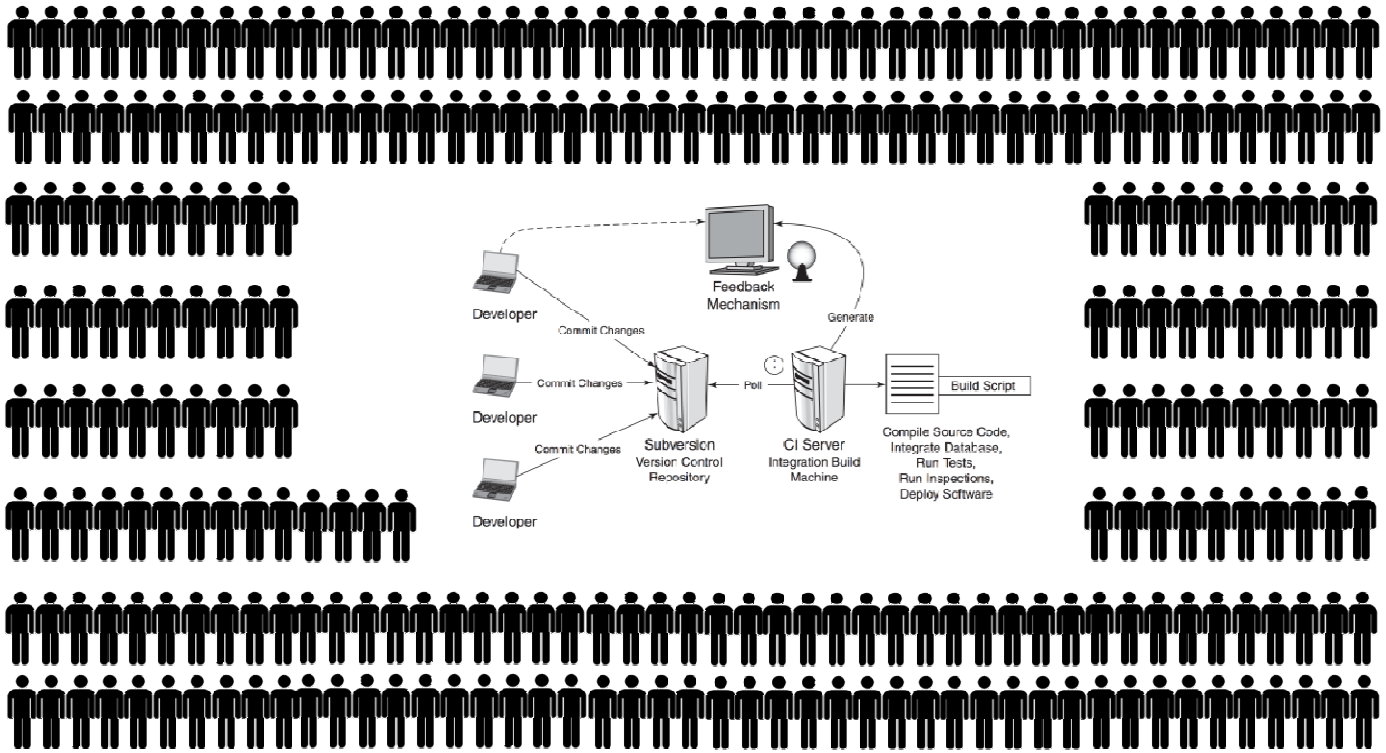
张燎原

2015-10-16

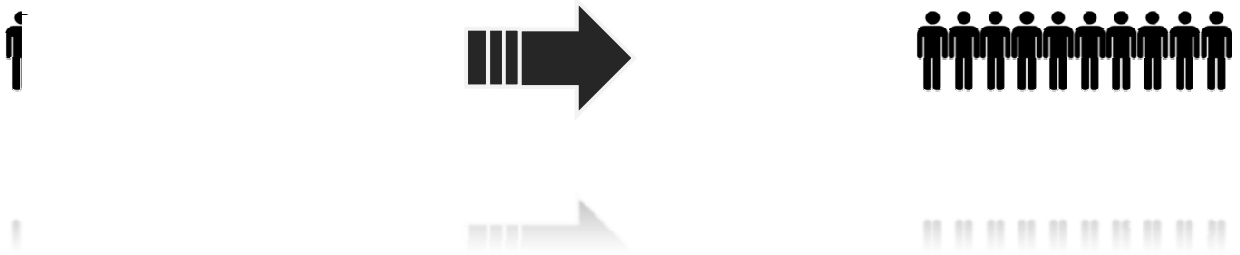
# 教科书上是这么说的…


Feedback
Mechanism
Developer
Commit Changes
Generate
Commit Changes
Developer
Poll
Build Script
Commit Changes
Subversion
Version Control
Repository
CI Server
Integration Build
Machine
Compile Source Code,
Integrate Database,
Run Tests,
Run Inspections,
Deploy Software
Developer

# 说好的大规模呢？

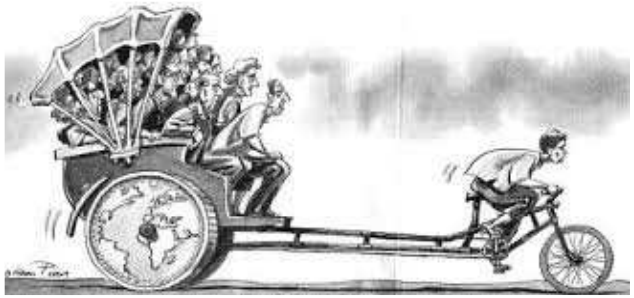

Feedback
Mechanism
Developer
Commit Changes
Generate
Commit Changes
Poll
Build Script
Developer
Commit Changes
Subversion
Version Control
Repository
CI Server
Integration Build
Machine
Compile Source Code,
Integrate Database,
Run Tests,
Run Inspections,
Deploy Software
Developer

# 持续集成支持团队

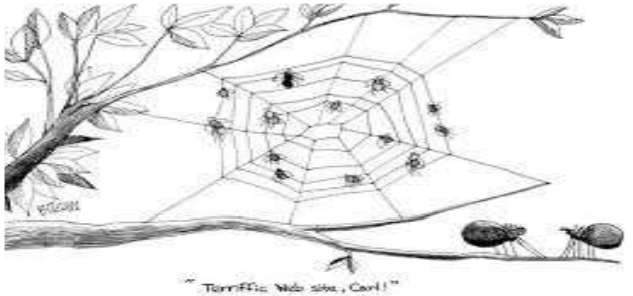


# 大规模持续集成的关键特性

规模大



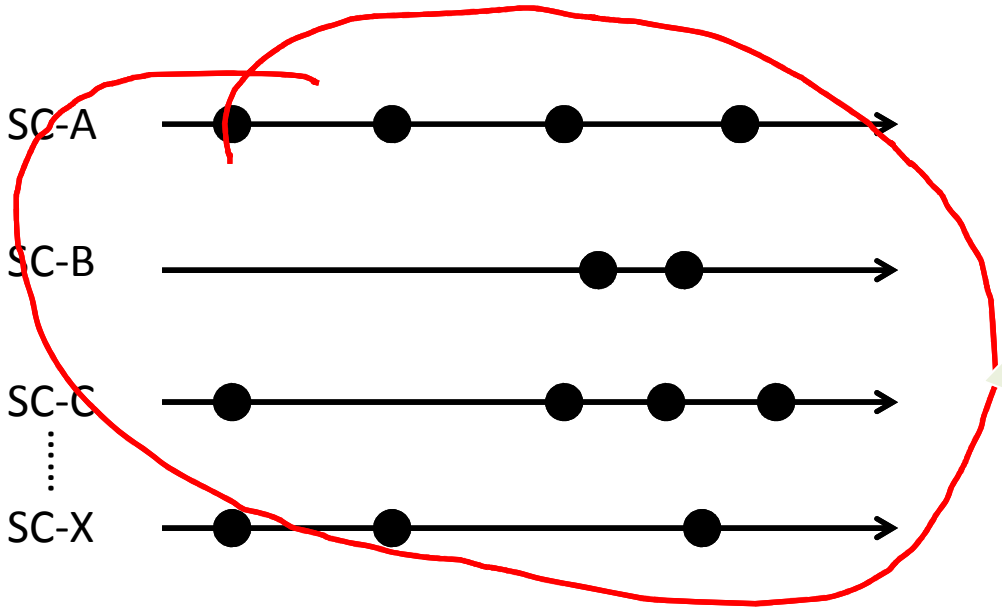
依赖多




影响广

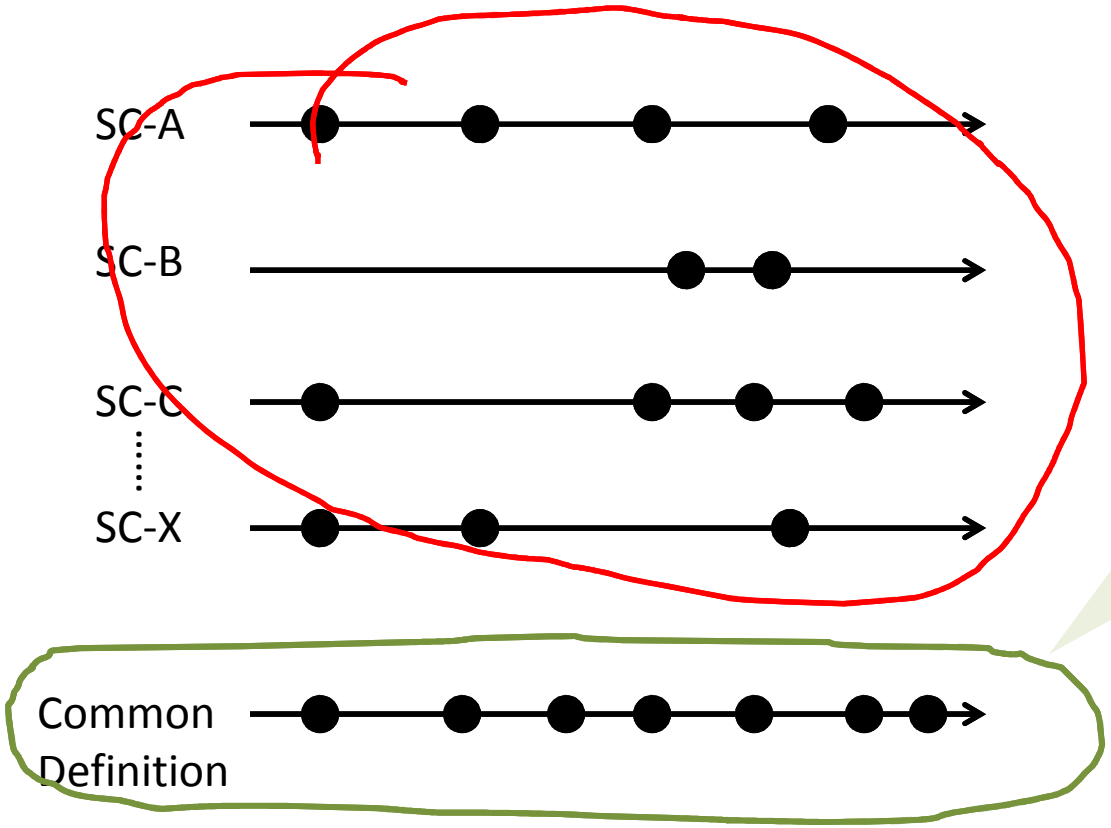

反馈长

# 规模大



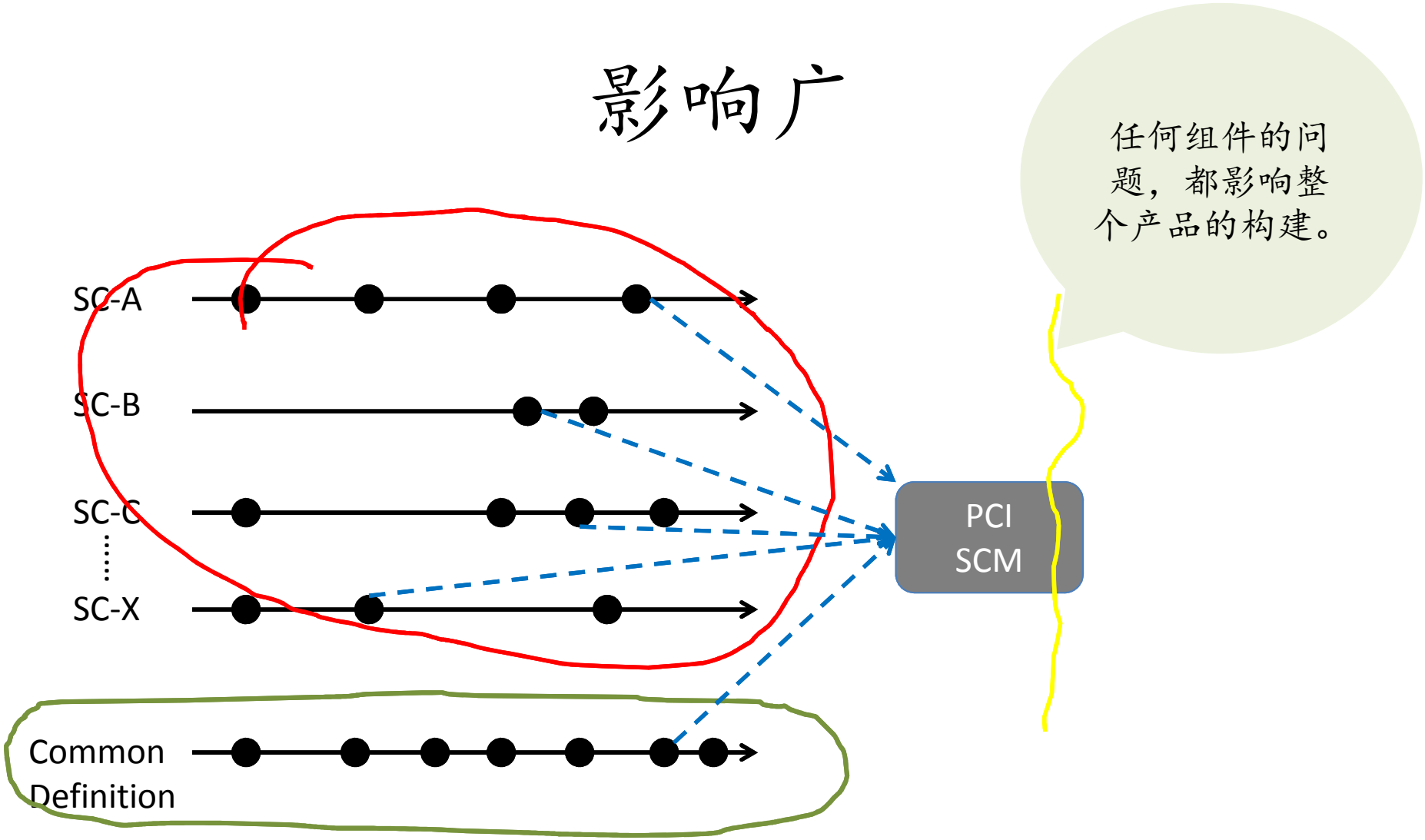

整个产品的构建由多组件构成，每个组件都有上百号人在贡献代码，整个产品的开发人员大于1000人。

# 依赖多

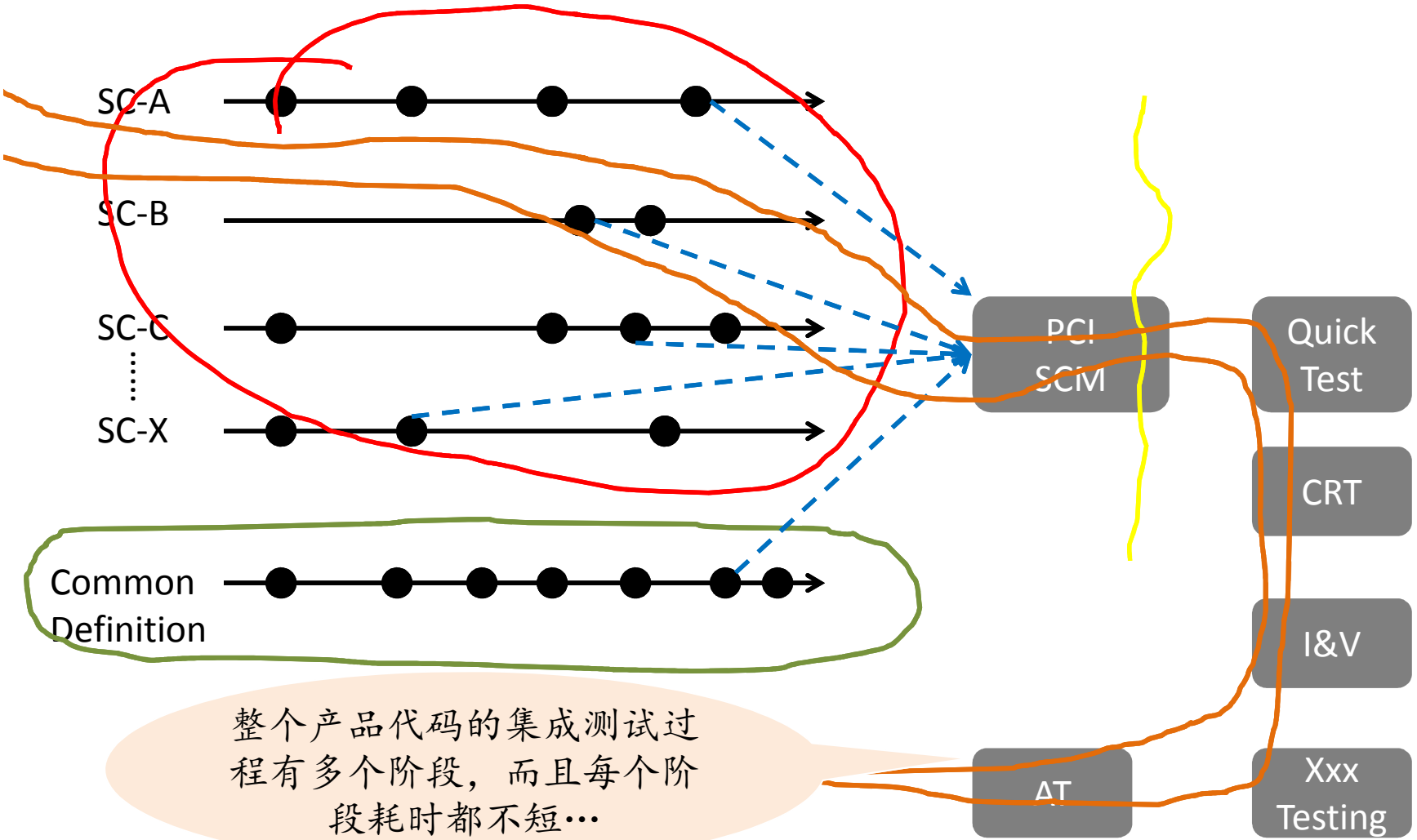



组件之间有比较多的依赖，而且有一个通用模块，定义了很多的通用消息、数据结构等，被所有的组件共享。

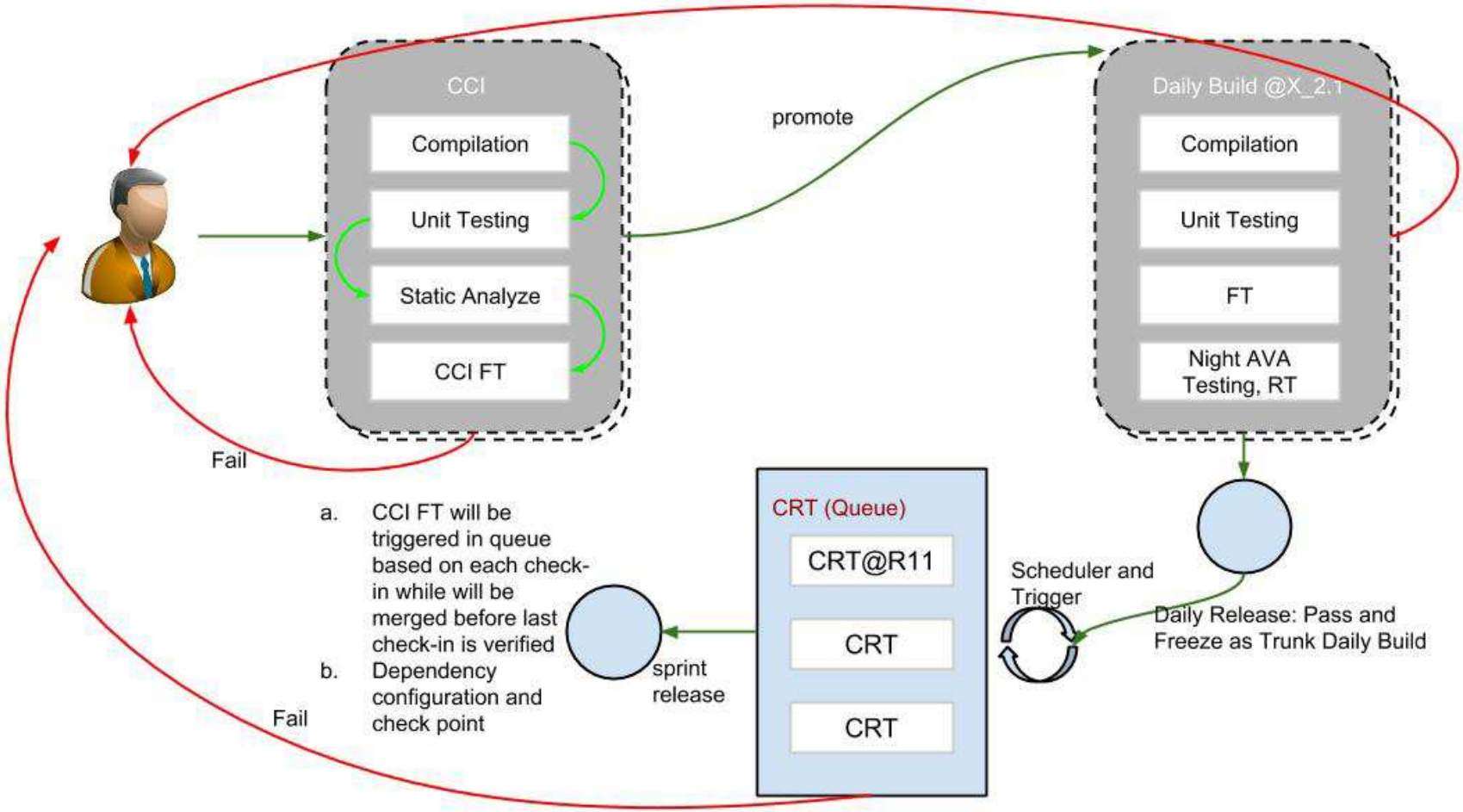
# 影响广


任何组件的问题，都影响整个产品的构建。
SC-A
SC-B
SC-C
SC-X
PCI
SCM
Common
Definition

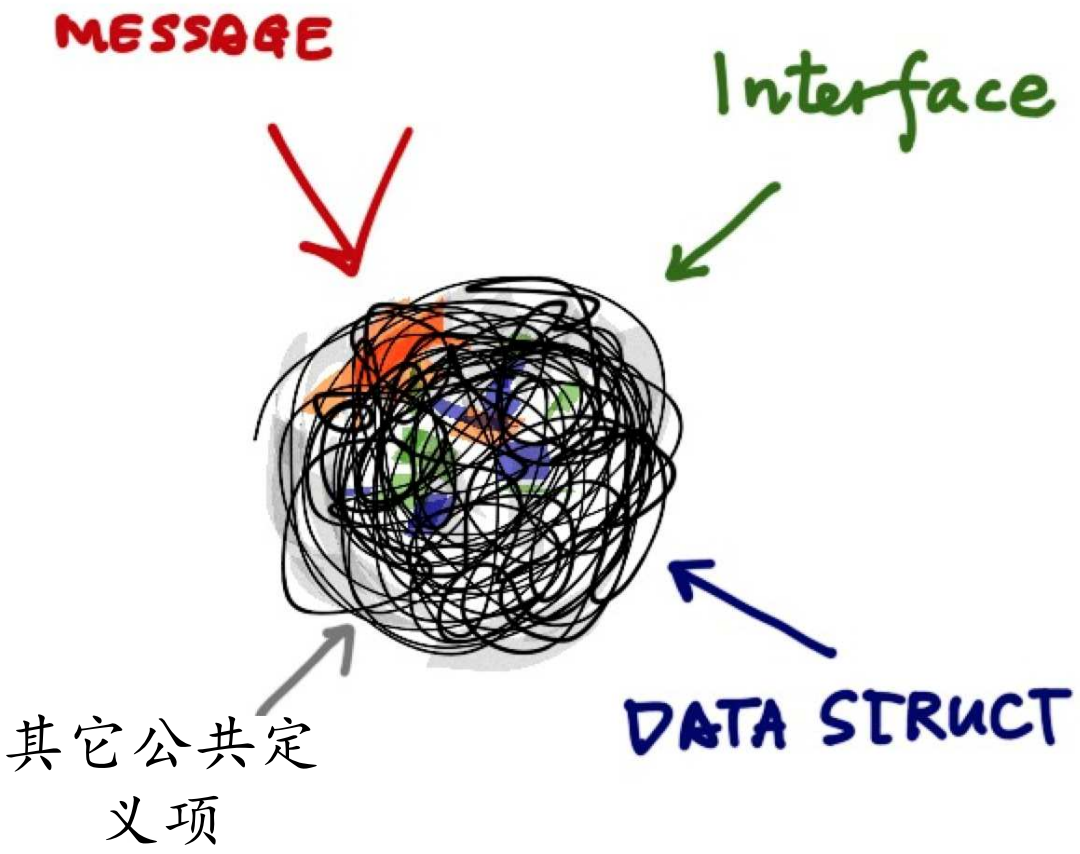
# 反馈长


SC-A
SC-B
SC-C
SC-X
Common Definition
PCI SCIM
Quick Test
CRT
I&V
AT
Xxx Testing
整个产品代码的集成测试过程有多个阶段，而且每个阶段耗时都不短…

# 持续集成流程是这个样子的



# 案例1：公共定义成了最大的依赖





图片来源：http://art.ce.cn/ylmb/ylpl/200810/16/t20081016_17082032.shtml

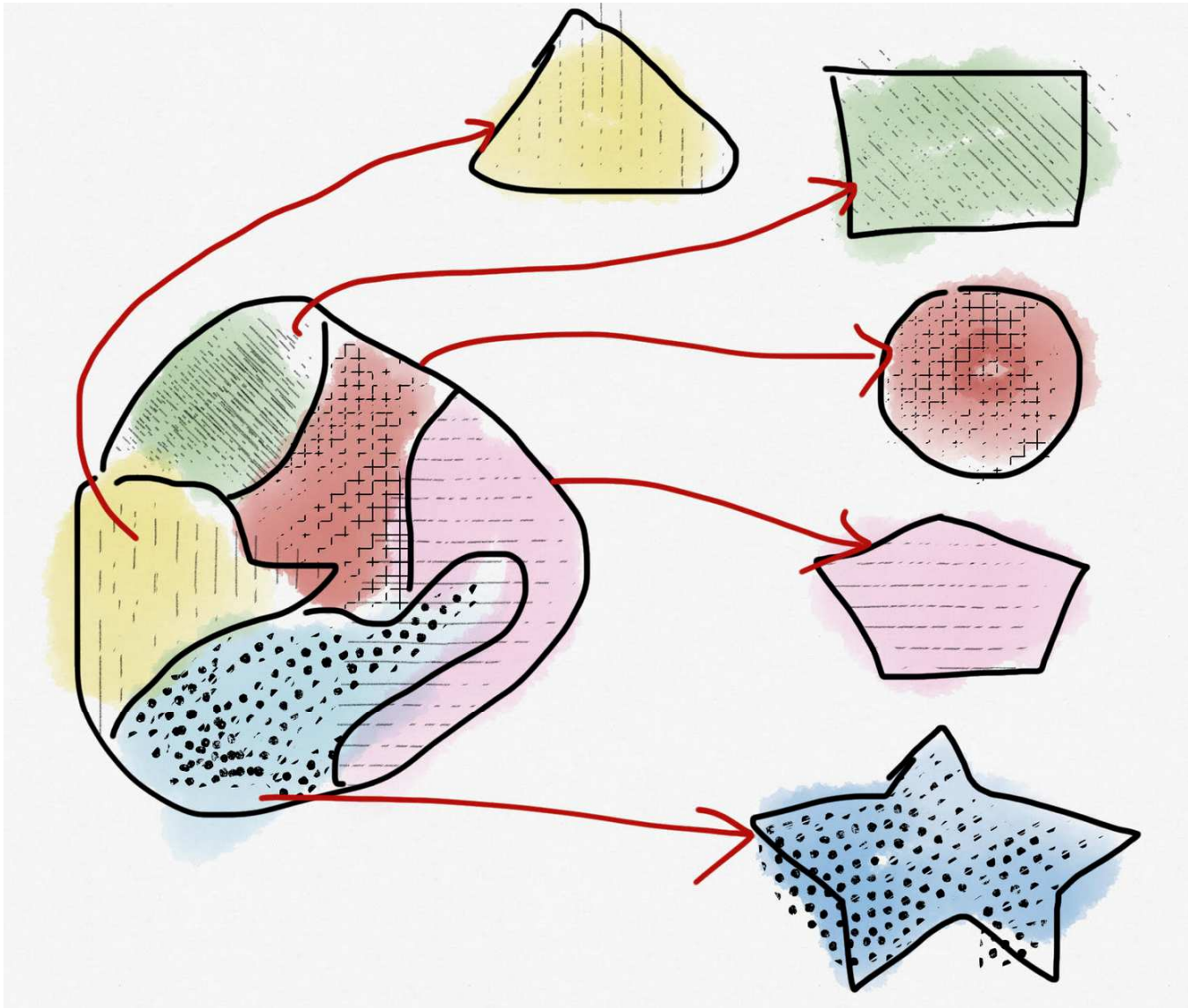
# 依赖管理(1)

| Common Definition | SC_A | SC_B | SC_C | SC_D | SC_E |
|---|---|---|---|---|---|
| Version 1.1 | +1 | +1 | +1 | +1 | +1 |
| Version 1.2 | +1 | +1 | +1 | +1 | +1 |
| Version 1.3 | +1 | +1 | +1 | +1 | +1 |
| Version 1.4 | +1 | +1 | +1 | +1 | +1 |
| Version 1.5 | +1 | +1 | +1 | +1 | +1 |
| Version 1.6 | +1 | +1 | +1 | +1 | +1 |
| Version 1.7 | +1 | +1 | +1 | +1 | +1 |

# 依赖管理(2)

| Common Definition | SC_A | SC_B | SC_C | SC_D | SC_E |
|---|---|---|---|---|---|
| Version 1.1 | No change | +1 | +1 | No change | No change |
| Version 1.2 | No change | +1 | No change | No change | No change |
| Version 1.3 | No change | No change | No change | +1 | No change |
| Version 1.4 | +1 | No change | No change | No change | No change |
| Version 1.5 | No change | No change | No change | No change | No change |
| Version 1.6 | No change | No change | +1 | No change | +1 |
| Version 1.7 | +1 | +1 | +1 | No change | +1 |

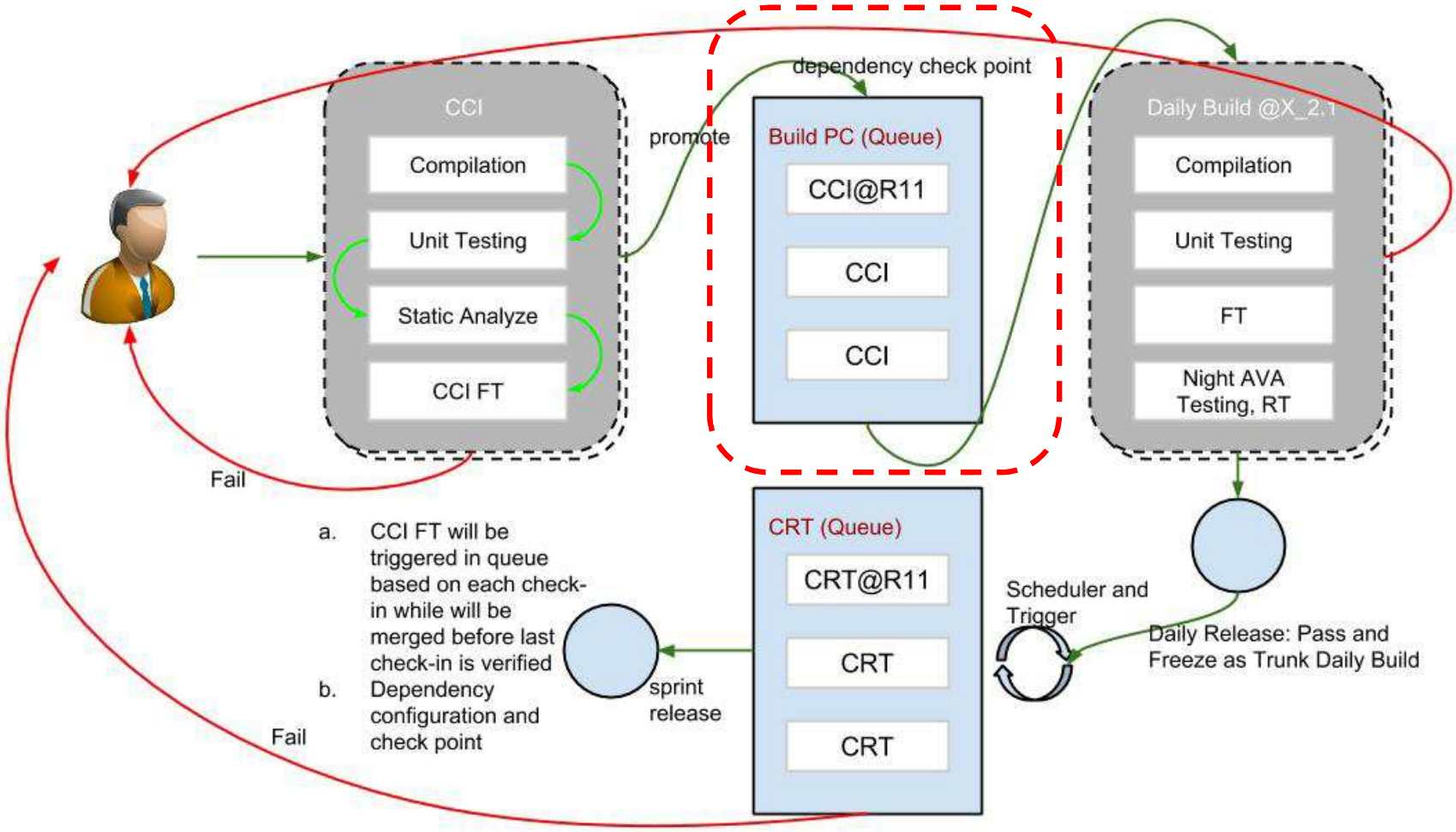
# 划分公共定义



# 依赖管理(3)

| Common Definition | | | | SC_A | SC_B | SC_C | SC_D |
|---|---|---|---|---|---|---|---|
| a | b+1 | c | d | +1 | | | |
| a | b | c | d | | | | |
| a | b+1 | c | d | +1 | | | |
| a | b | c+1 | d | | +1 | | |
| a | b | c | d+1 | | +1 | +1 | +1 |
| a+1 | b | c+1 | d | +1 | +1 | | |
| a | b+1 | c | d+1 | +1 | +1 | +1 | +1 |

# 然后持续集成流程是这个样子的



# 案例2：Cherry Picking

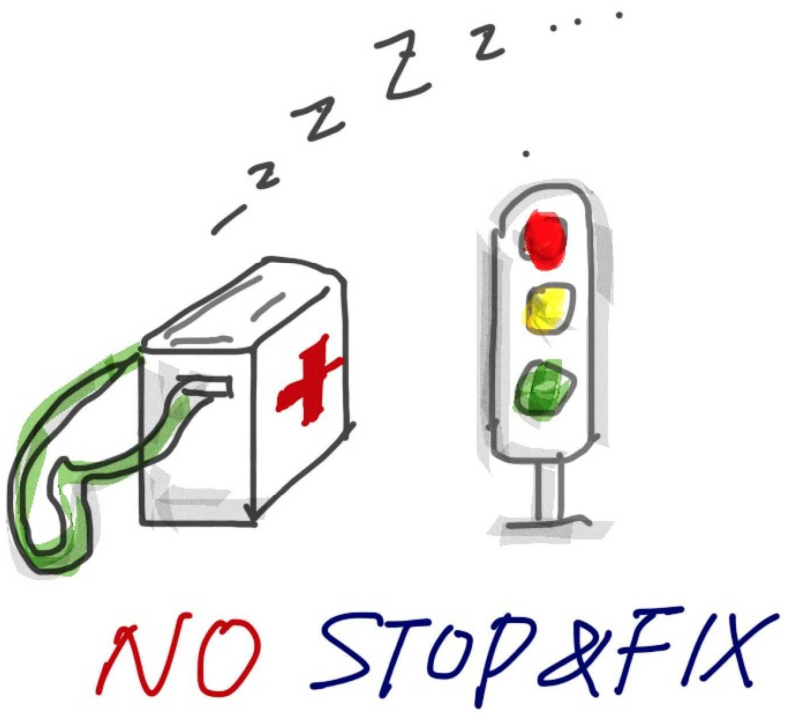
# 理由1：每个组件的交付质量偏差


前期工作处在赶工的状态，普遍认为写的代码越快越多越好，放进了版本库里就是开发工作的DONE。

# 理由2：持续集成下游等待出包的时间过长

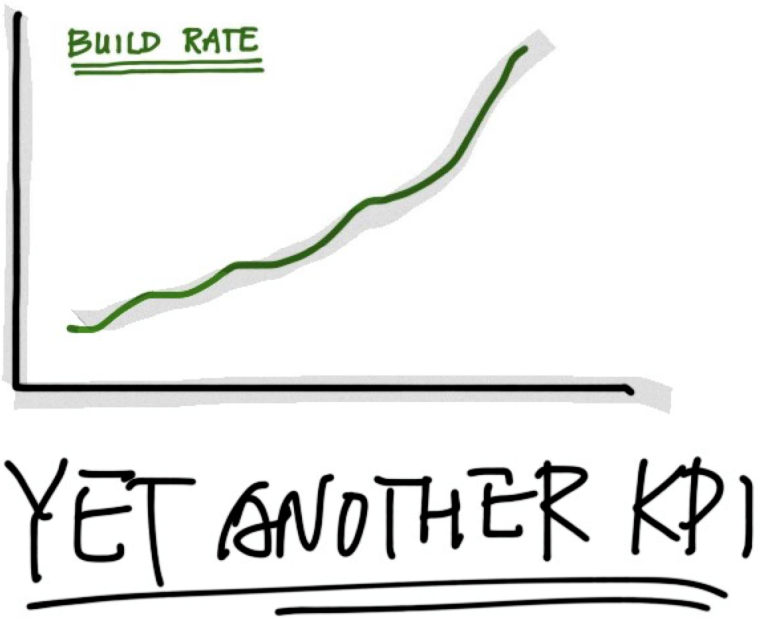

由于前期代码质量的原因，导致一致成功的集成消耗时间过长，持续集成的下游无活可做，催SCM赶紧给个可工作的包用于下游的工作安排。

# 理由3：不愿意承受任何的STOP

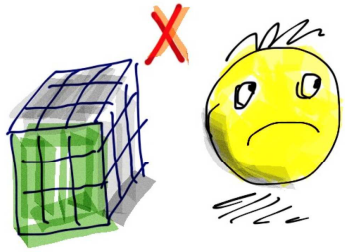



当产品在持续集成系统中验证出问题，影响是所有组件，做为某一个组件的Owner，不愿意STOP所有的检入，因为“我”没有问题。

# 理由4：编包人员的KPI是出包率

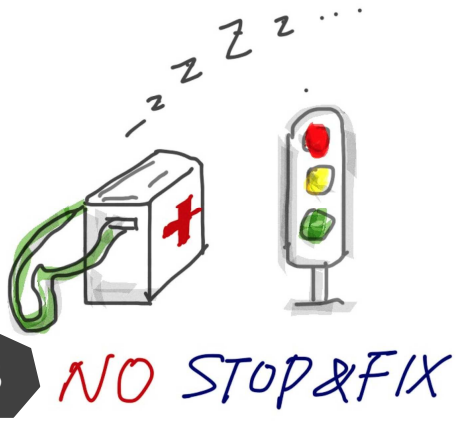



给SCM的同事定义了KPI，即出包的频率是多少？越多越好，越成功越好。

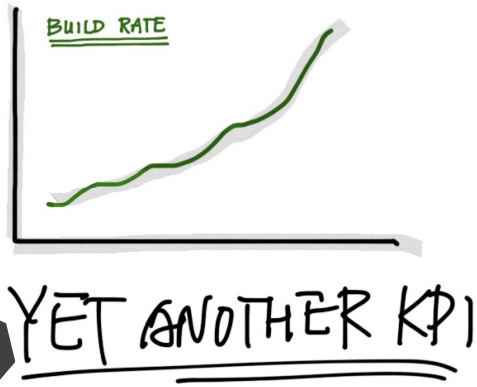



每个组件的交付质量直接影响是否能出包，且组件质量编差


持续集成的下游等待出包的时间过长。

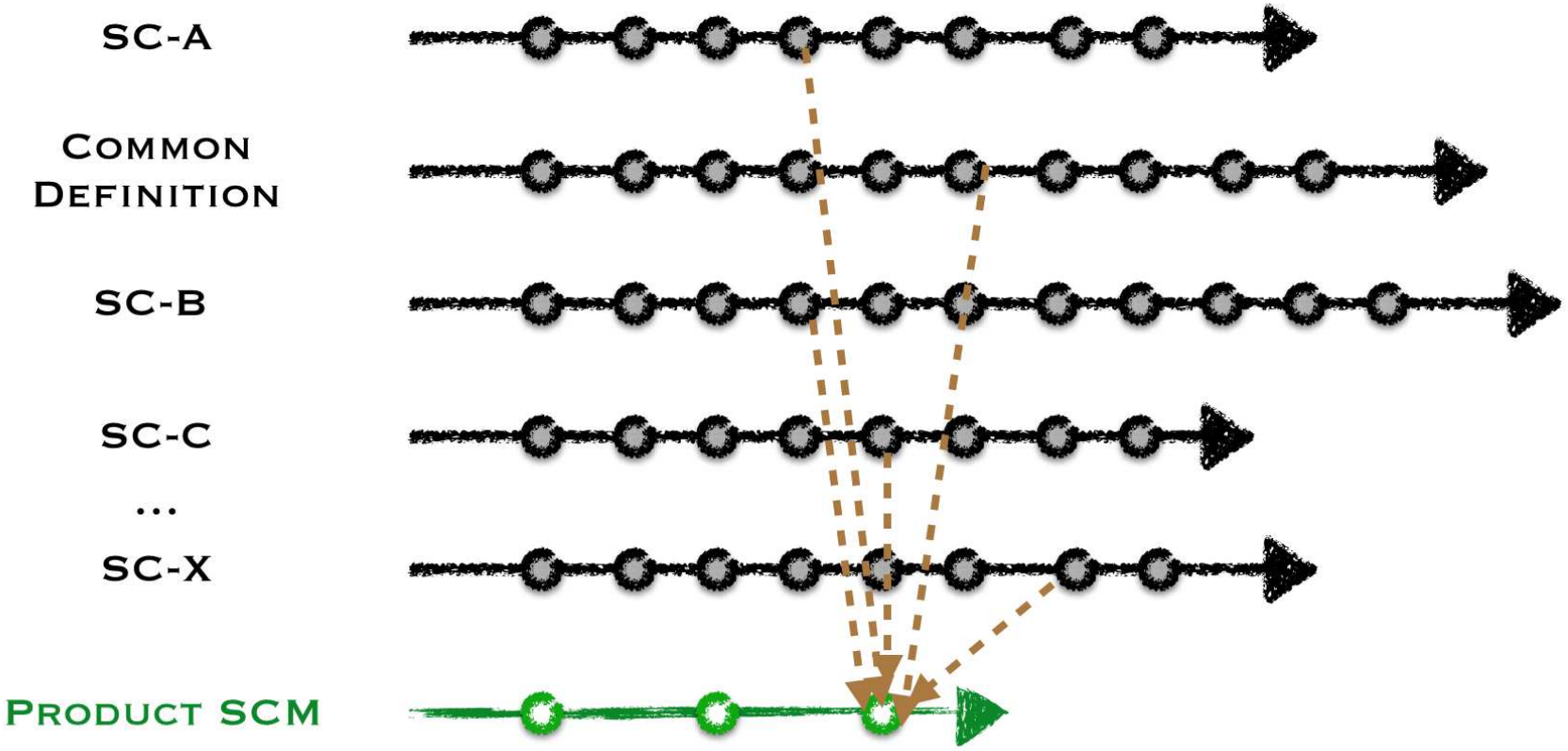



不愿意承担STOP的开销成本




对编包人员的考核是出包率

# 为了应对这些问题，所以这样

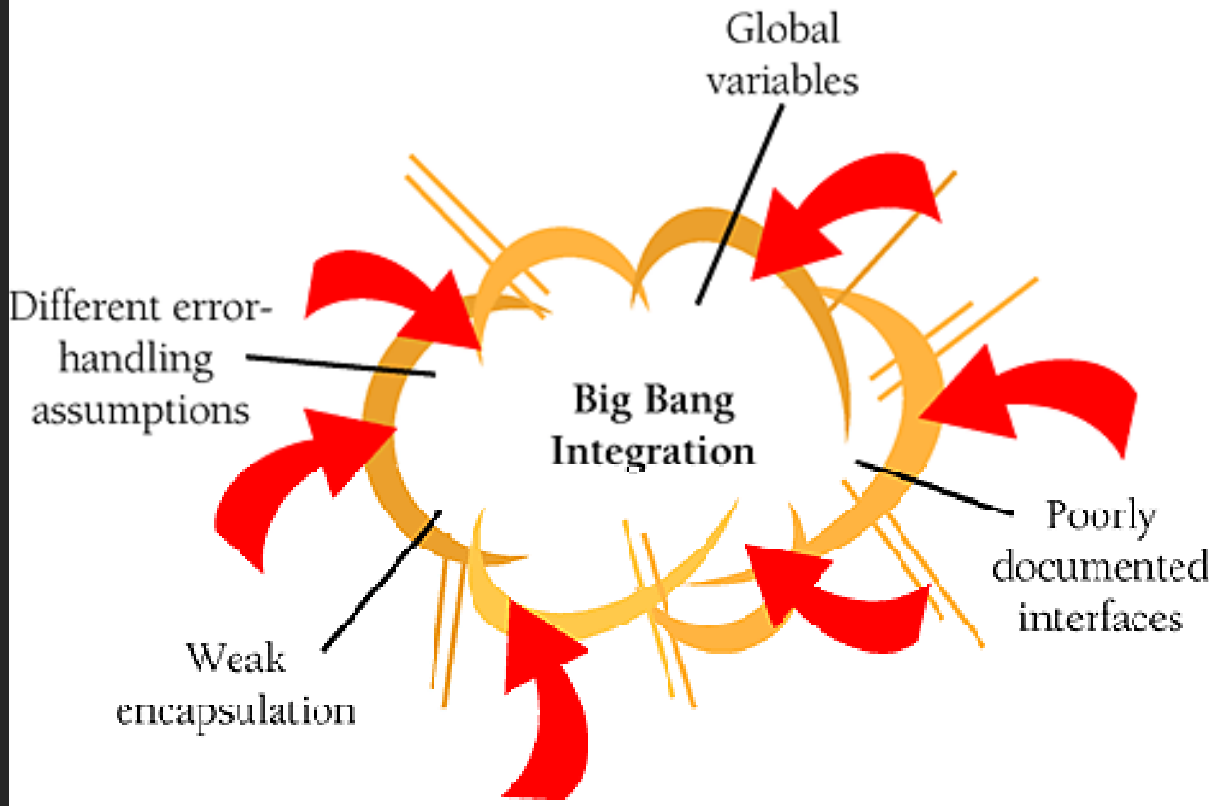


结果真的很糟糕……

制造问题的速度远远超过解决问题的速度：每个组件只关注自己往前开发，而验证总是滞后很久。

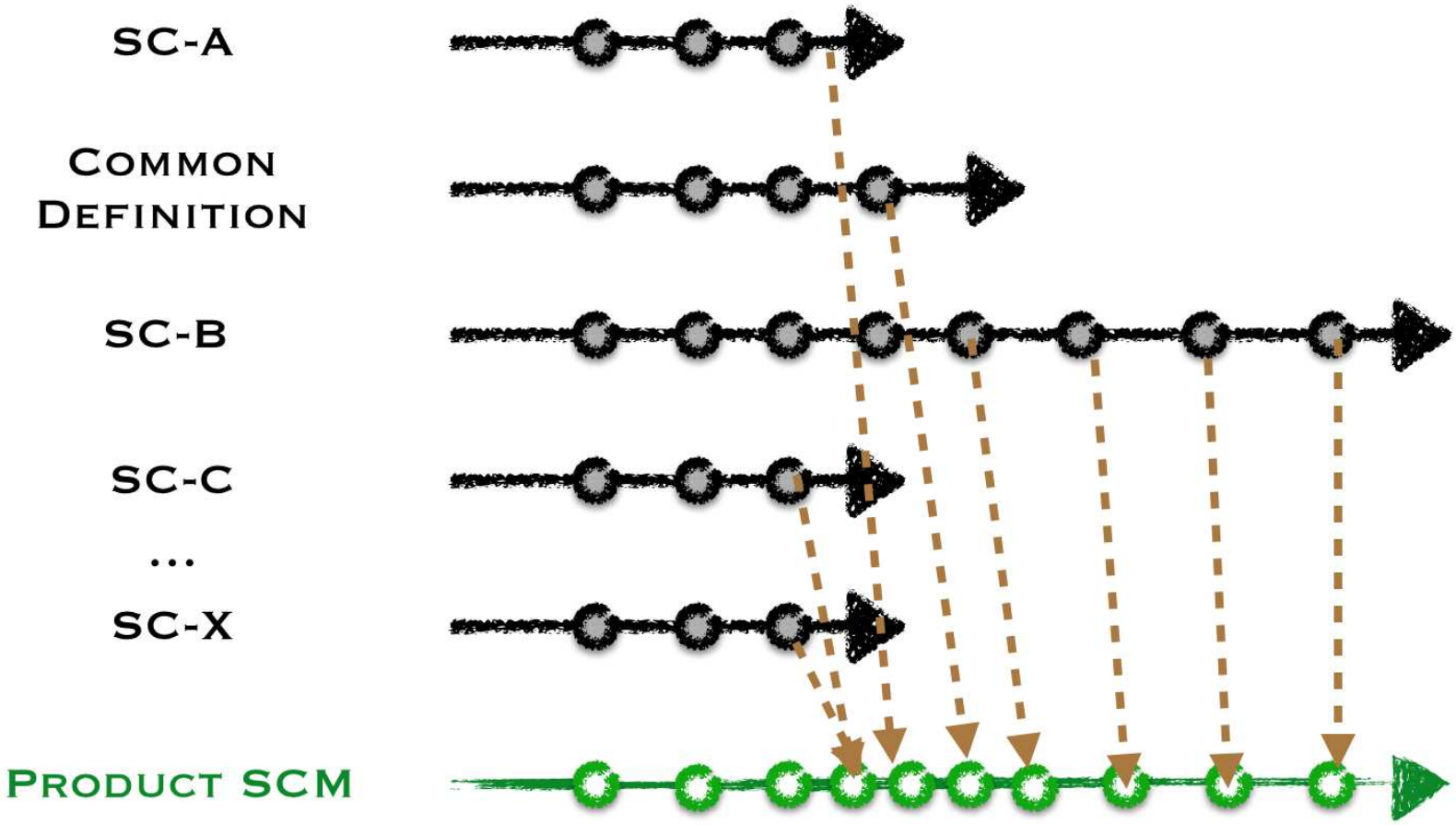
各个组件的开发基于的代码未在产品版本上验证过，Codebase越来越糟糕，每个组件的开发越来越没有信心。

每次代码的pick总是会引来大量组件代码的集成编译，因为开发的速度大于集成的速度，集成的粒度越大，集成的风险就越高，最后演变成一次次的Big Bang集成。



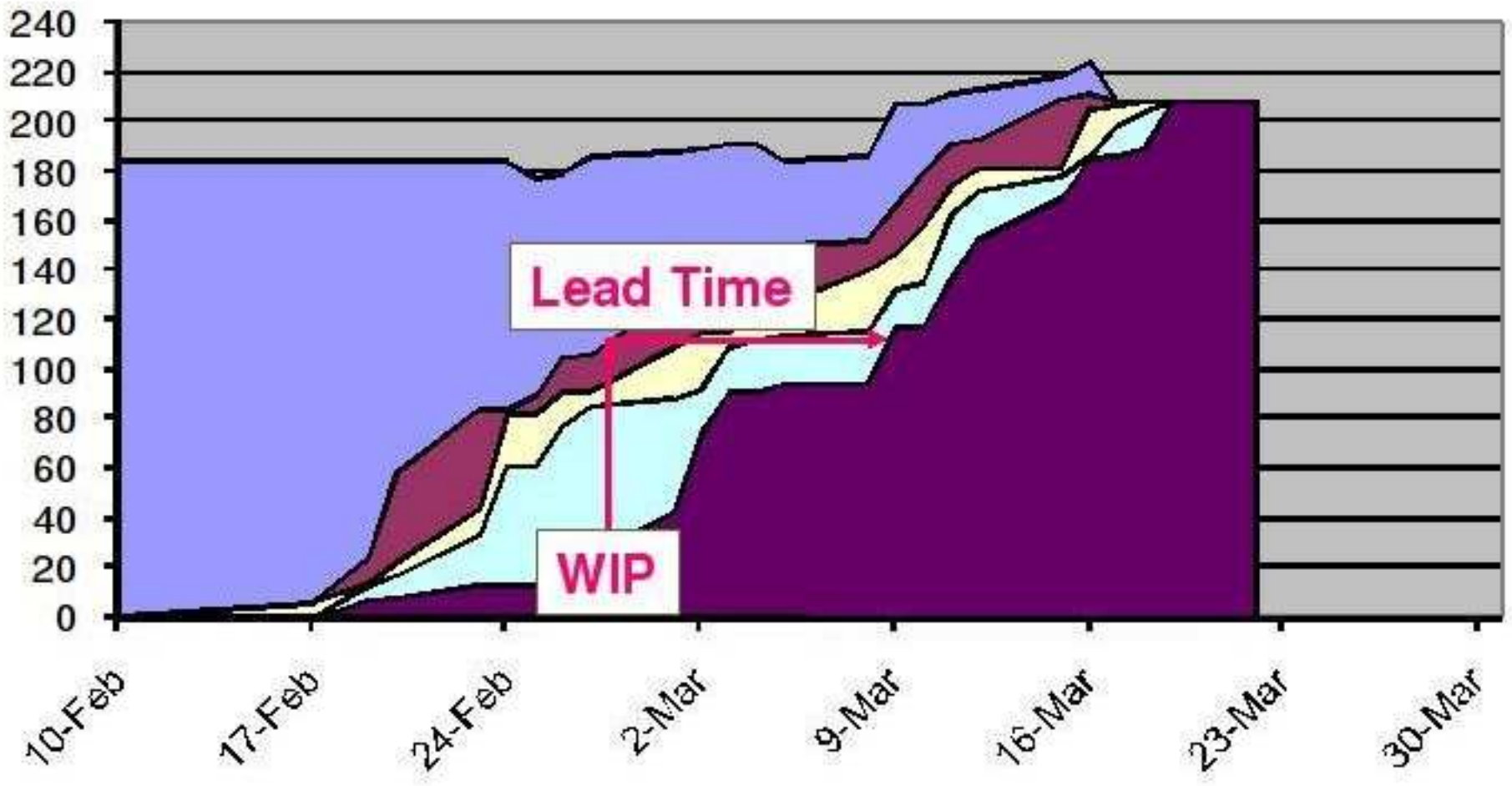

因为产品验证的进度远远落后于开发的进度，很多时候，验证与开发的关注点会有比较大的时间跨度，更多时候成了“鸡同鸭讲”

# 如果这样就好了，:p

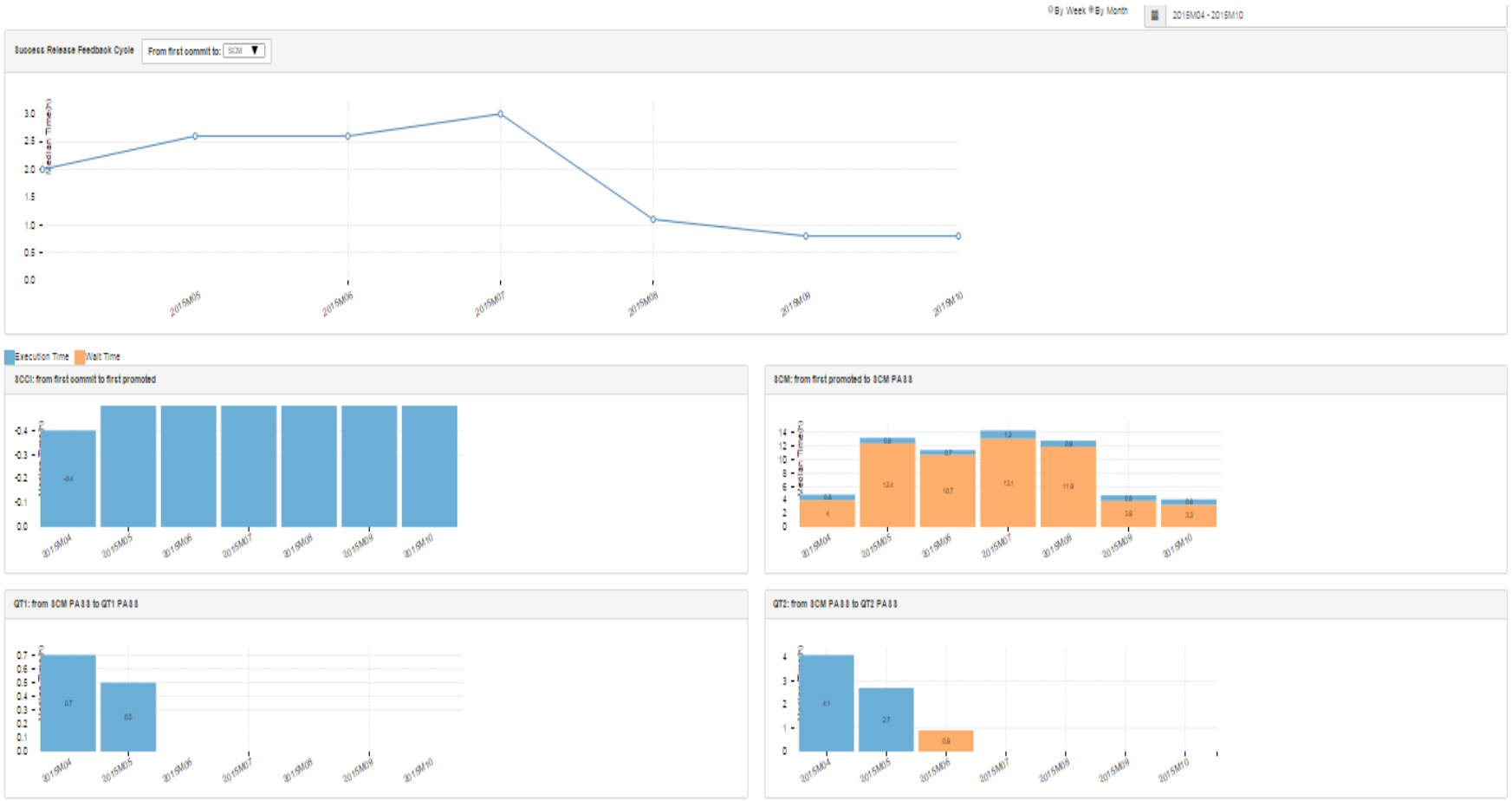


# STOP & FIX, Rollback & Quick Fix

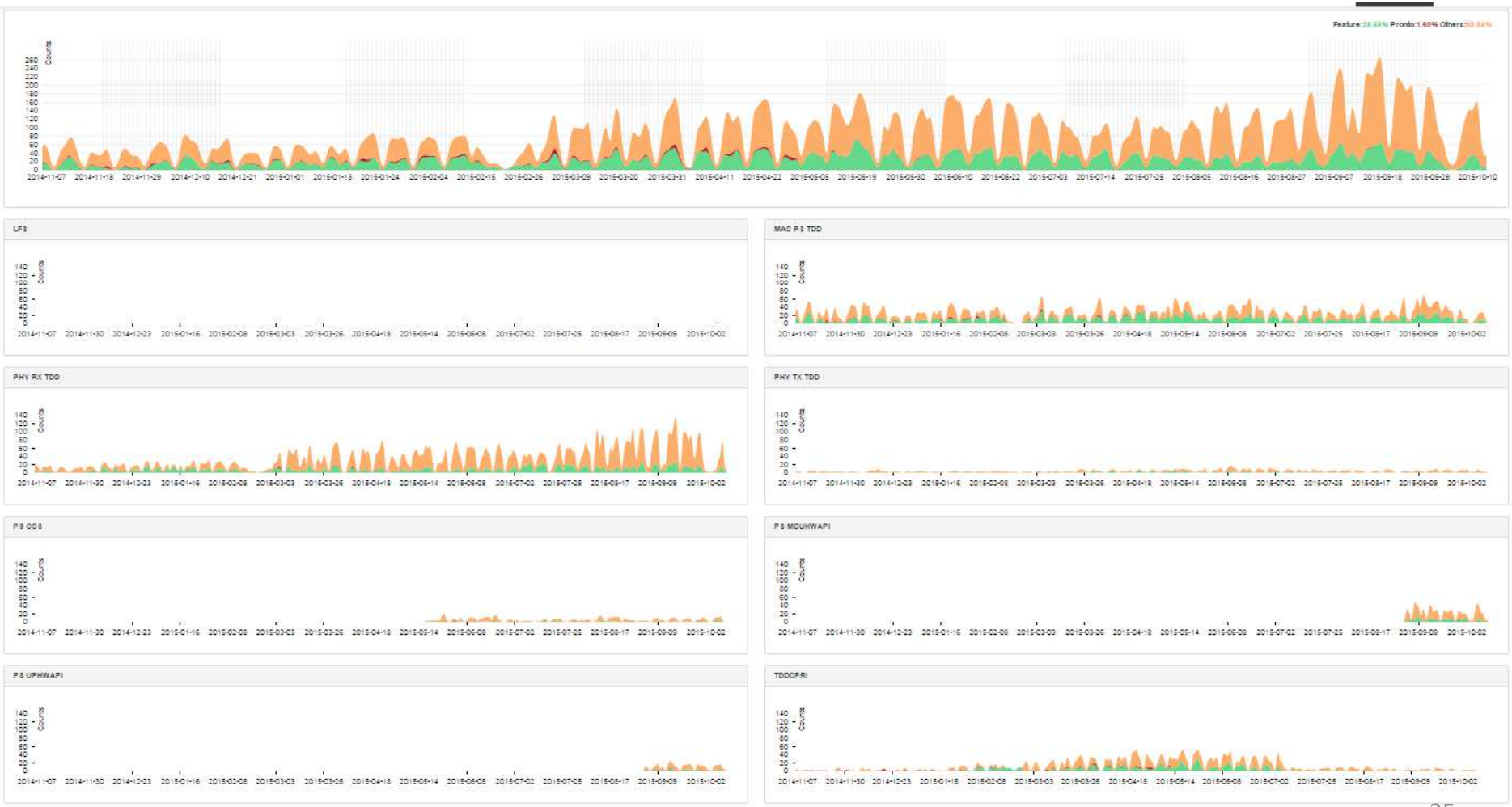
# 定义正确的度量


240
220
200
180
160
140
120
100
80
60
40
20
0
Lead Time
WIP
10-Feb
17-Feb
24-Feb
2-Mar
9-Mar
16-Mar
23-Mar
30-Mar

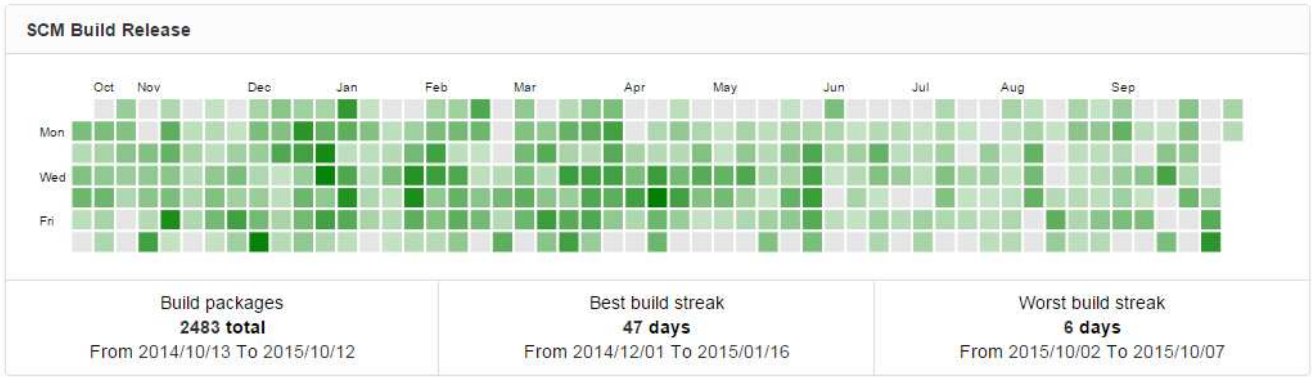
# 确定在不同阶段的Cycle Time

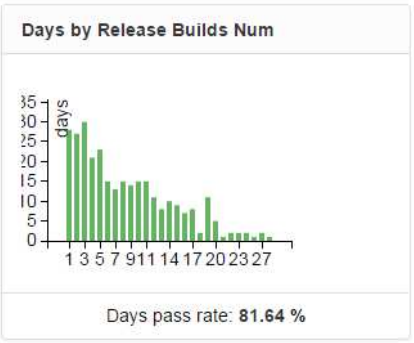


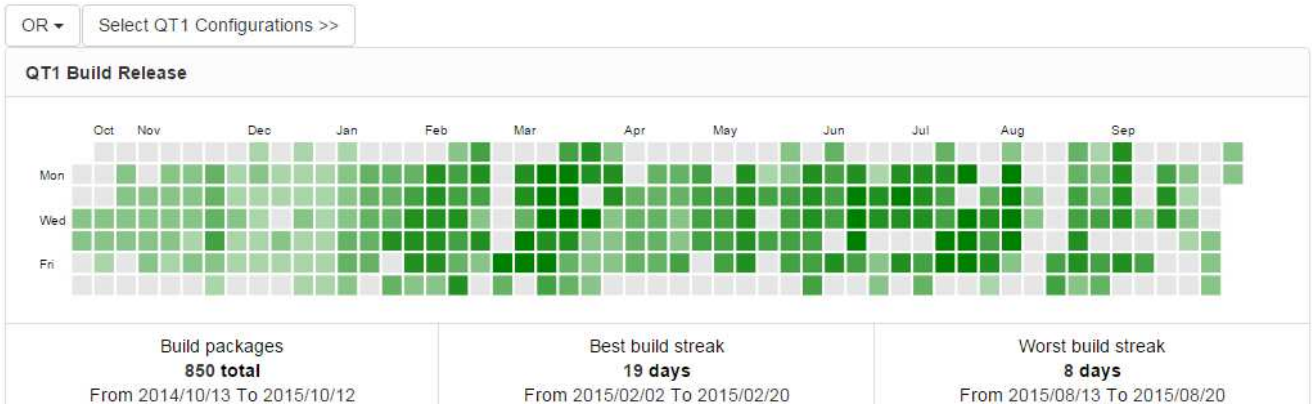
# 确定各组件的集成代码提交频率和集成频率

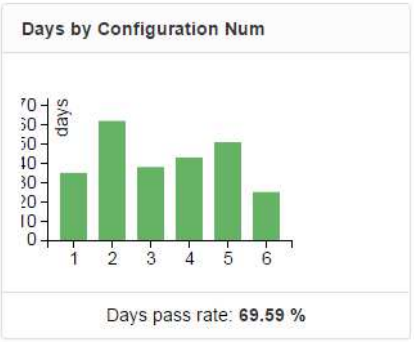


# 确定成功编包的频率和验证频率

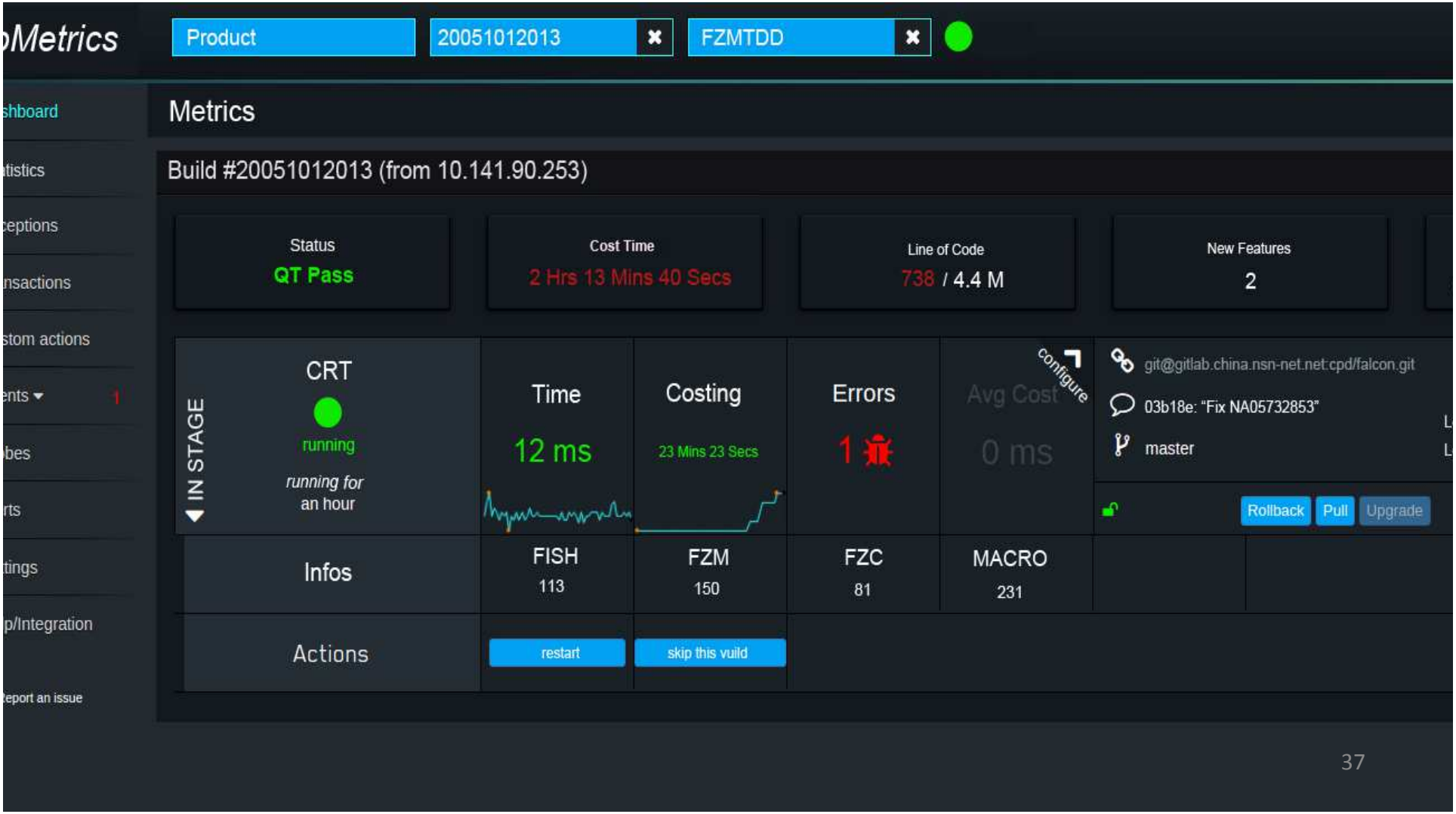


# 现在的系统是这个样子的



It’s time to say

# THANK U FOR YOUR LISTENING

blog.zhangliaoyuan.com

@zhangliaoyuan