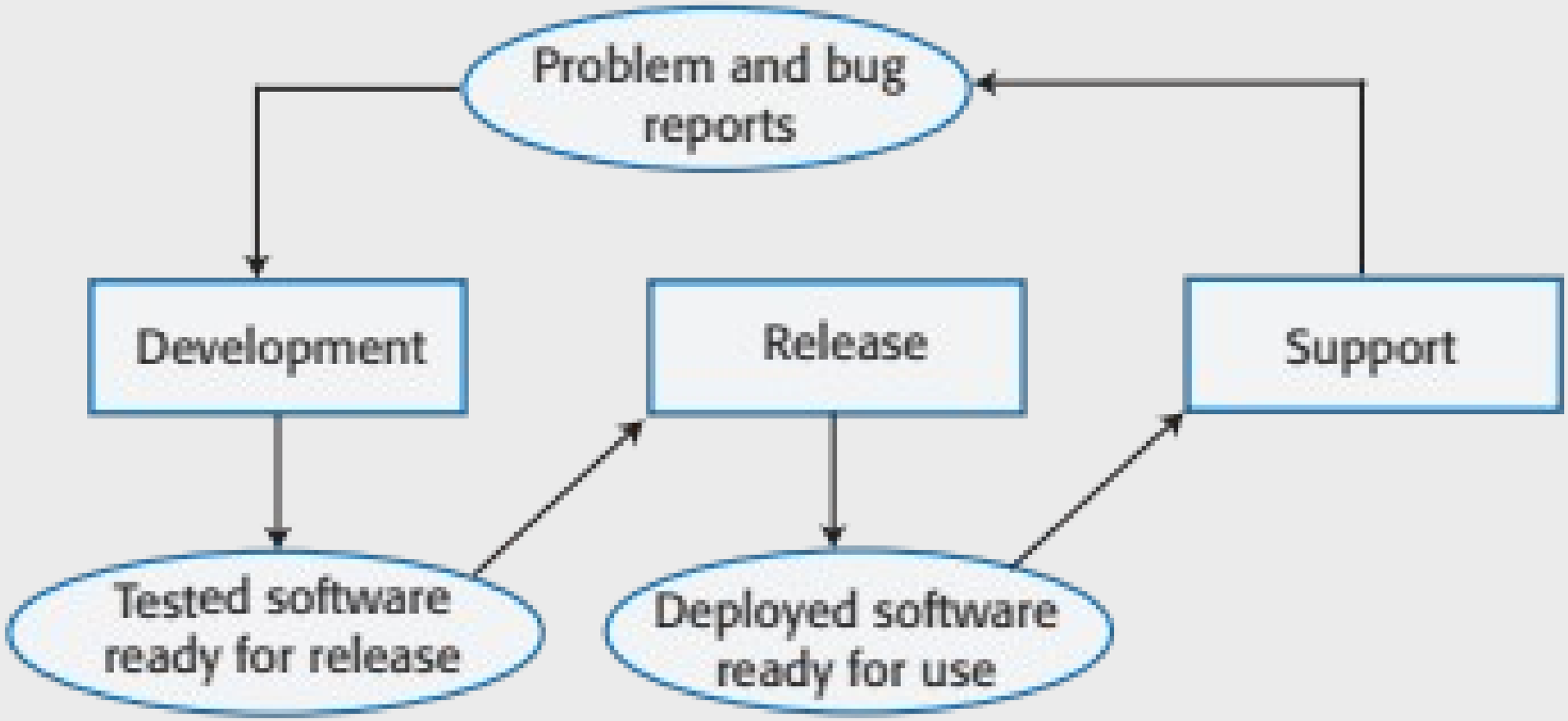
DevOps and Code Management

Software support

- Traditionally, separate teams were responsible software development, software release and software support.
- The development team passed over a ‘final’ version of the software to a release team. This team then built a release version, tested this and prepared release documentation before releasing the software to customers.
- A third team was responsible for providing customer support.
  - The original development team were sometimes also responsible for implementing software changes.
  - Alternatively, the software may have been maintained by a separate ‘maintenance team’.

Figure 10.1 Development, release and support



DevOps

- There are inevitable delays and overheads in the traditional support model.
- To speed up the release and support processes, an alternative approach called DevOps (Development+Operations) has been developed.
- Three factors led to the development and widespread adoption of DevOps:
  - Agile software engineering reduced the development time for software, but the traditional release process introduced a bottleneck between development and deployment.
  - Amazon re-engineered their software around services and introduced an approach in which a service was developed and supported by the same team. Amazon's claim that this led to significant improvements in reliability was widely publicized.
  - It became possible to release software as a service, running on a public or private cloud. Software products did not have to be released to users on physical media or downloads.

Figure 10.2 DevOps

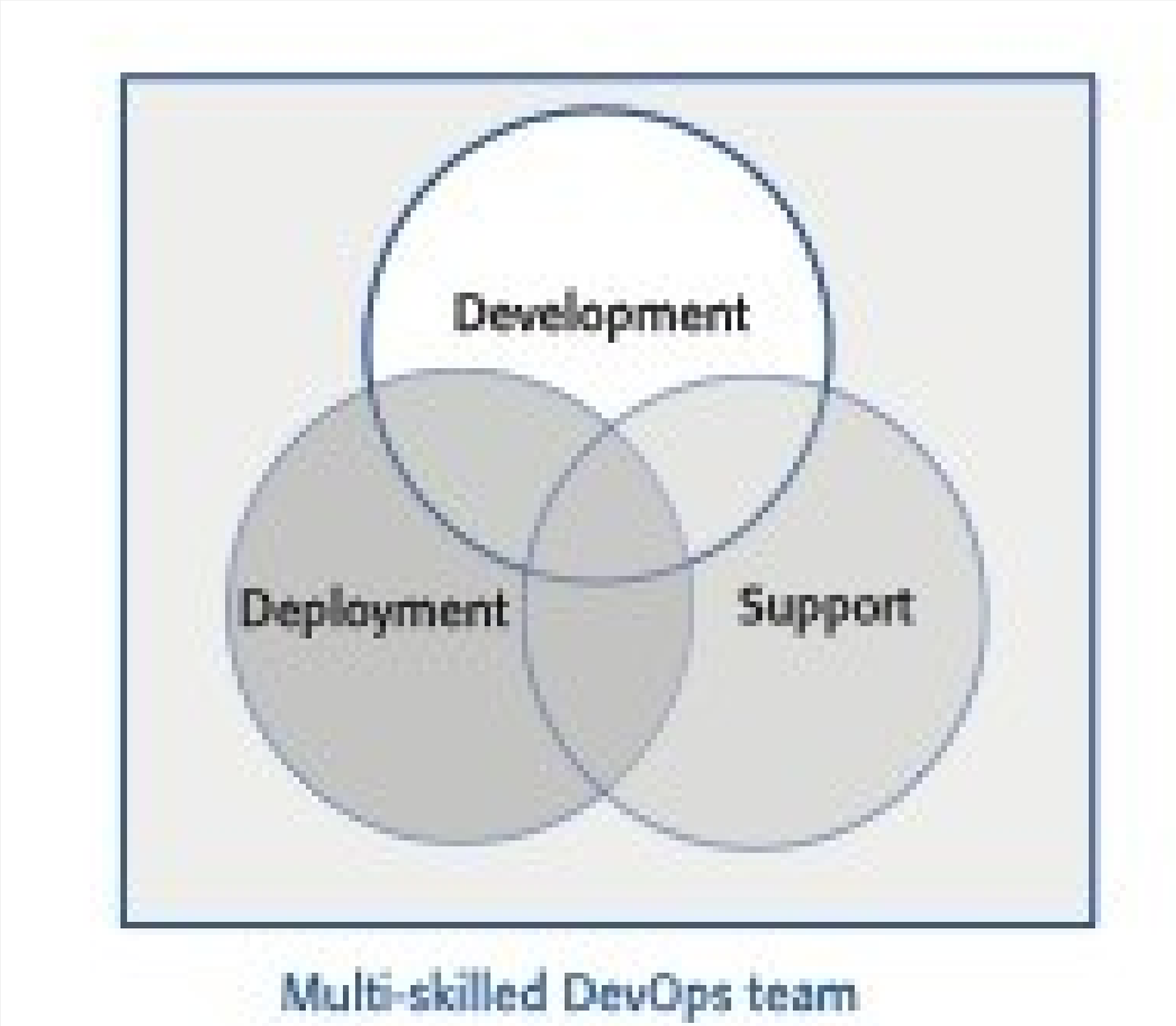


Table 10.1 DevOps principles

Everyone is responsible for everything
All team members have joint responsibility for developing, delivering and supporting the software.

Everything that can be automated should be automated
All activities involved in testing, deployment and support should be automated if it is possible to do so. There should be mimimal manual involvement in deploying software.

Measure first, change later
DevOps should be driven by a measurement program where you collect data about the system and its operation. You then use the collected data to inform decisions about changing DevOps processes and tools.

Table 10.2 Benefits of DevOps

Faster deployment
Software can be deployed to production more quickly because communication delays between the people involved in the process are dramatically reduced.

Reduced risk
The increment of functionality in each release is small so there is less chance of feature interactions and other changes causing system failures and outages.

Faster repair
DevOps teams work together to get the software up and running again as soon as possible. There is no need to discover which team were responsible for the problem and to wait for them to fix it.

More productive teams
DevOps teams are happier and more productive than the teams involved in the separate activities. Because team members are happier, they are less likely to leave to find jobs elsewhere.

Code management

- During the development of a software product, the development team will probably create tens of thousands of lines of code and automated tests.
- These will be organized into hundreds of files. Dozens of libraries may be used, and several, different programs may be involved in creating and running the code.
- Code management is a set of software-supported practices that is used to manage an evolving codebase.
- You need code management to ensure that changes made by different developers do not interfere with each other, and to create different product versions.
- Code management tools make it easy to create an executable product from its source code files and to run automated tests on that product.

Table 10.3 A code management problem

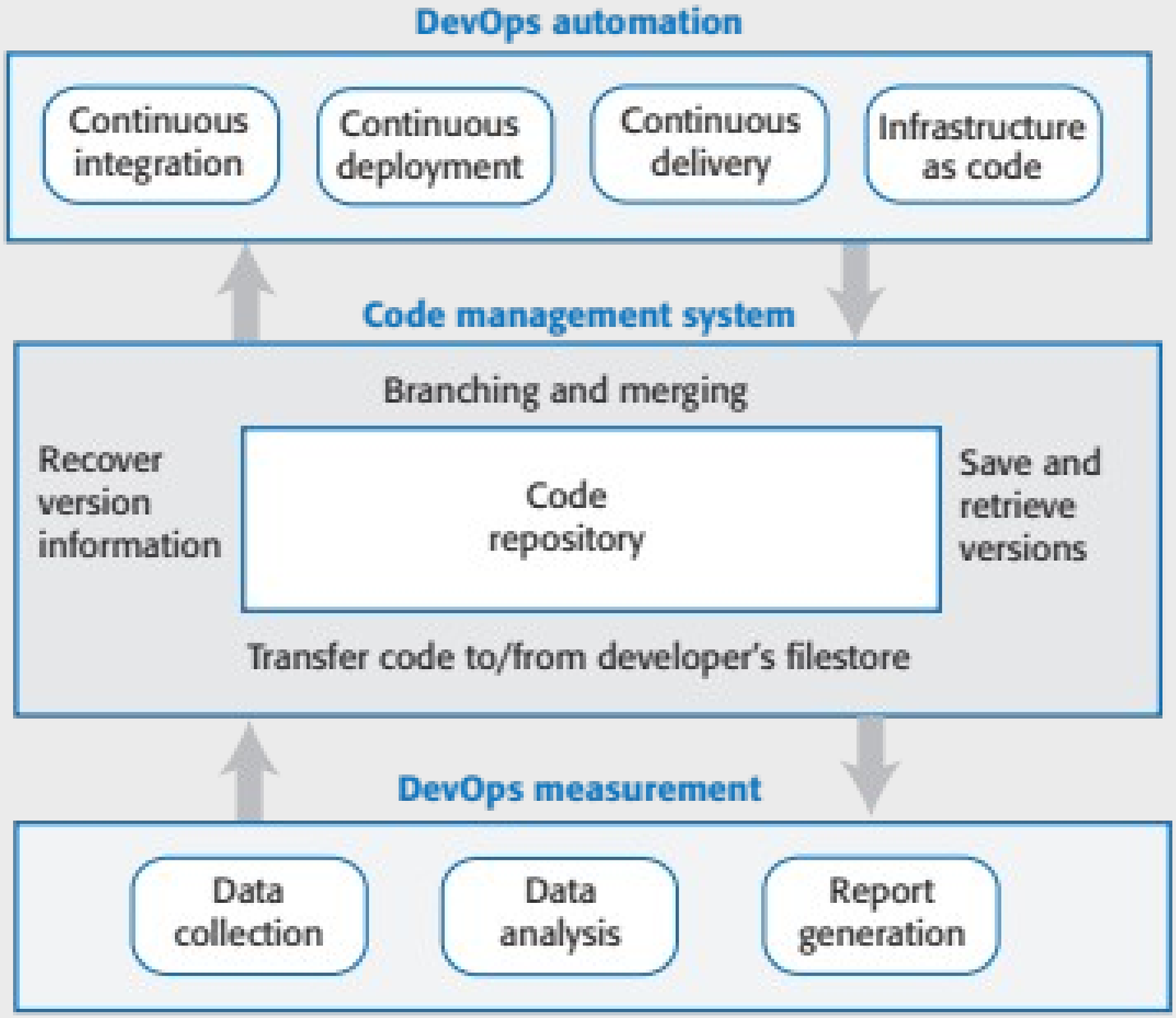
Alice and Bob worked for a company called FinanceMadeSimple and were team members involved in developing a personal finance product. Alice discovered a bug in a module called TaxReturnPreparation. The bug was that a tax return was reported as filed but, sometimes, it was not actually sent to the tax office. She edited the module to fix the bug. Bob was working on the user interface for the system and was also working on TaxReturnPreparation. Unfortunately, he took a copy before Alice had fixed the bug and, after making his changes, he saved the module. This overwrote Alice's changes but she was not aware of this.

The product tests did not reveal the bug as it was an intermittent failure that depended on the sections of the tax return form that had been completed. The product was launched with the bug. For most users, everything worked OK. However, for a small number of users, their tax returns were not filed and they were fined by the revenue service. The subsequent investigation showed the software company was negligent. This was widely publicised and, as well as a fine from the tax authorities, users lost confidence in the software. Many switched to a rival product. FinanceMade Simple failed and both Bob and Alice lost their jobs.

Code management and DevOps

- Source code management, combined with automated system building, is essential for professional software engineering.
- In companies that use DevOps, a modern code management system is a fundamental requirement for ‘automating everything’.
- Not only does it store the project code that is ultimately deployed, it also stores all other information that is used in DevOps processes.
- DevOps automation and measurement tools all interact with the code management system

Figure 10.3 Code management and Devops



Code management fundamentals

- Code management systems provide a set of features that support four general areas:
  - **Code transfer** Developers take code into their personal file store to work on it then return it to the shared code management system.
  - **Version storage and retrieval** Files may be stored in several different versions and specific versions of these files can be retrieved.
  - **Merging and branching** Parallel development branches may be created for concurrent working. Changes made by developers in different branches may be merged.
  - **Version information** Information about the different versions maintained in the system may be stored and retrieved

Code repository

- All source code management systems have the general form shown in Figure 10.3. with a shared repository and a set of features to manage the files in that repository:
  - All source code files and file versions are stored in the repository, as are other artefacts such as configuration files, build scripts, shared libraries and versions of tools used.
  - The repository includes a database of information about the stored files such as version information, information about who has changed the files, what changes were made at what times, and so on.
- Files can be transferred to and from the repository and information about the different versions of files and their relationships may be updated.
  - Specific versions of files and information about these versions can always be retrieved from the repository.

Table 10.4 Features of code management systems

Version and release identification
Managed versions of a code file are uniquely identified when they are submitted to the system and can be retrieved using their identifier and other file attributes.

Change history recording
The reasons why changes to a code file have been made are recorded and maintained.

Independent development
Several developers can work on the same code file at the same time. When this is submitted to the code management system, a new version is created so that files are never overwritten by later changes.

Project support
All of the files associated with a project may be checked out at the same time.There is no need to check out files one at a time.

Storage management
The code management system includes efficient storage mechanisms so that it doesn’t keep multiple copies of files that have only small differences.

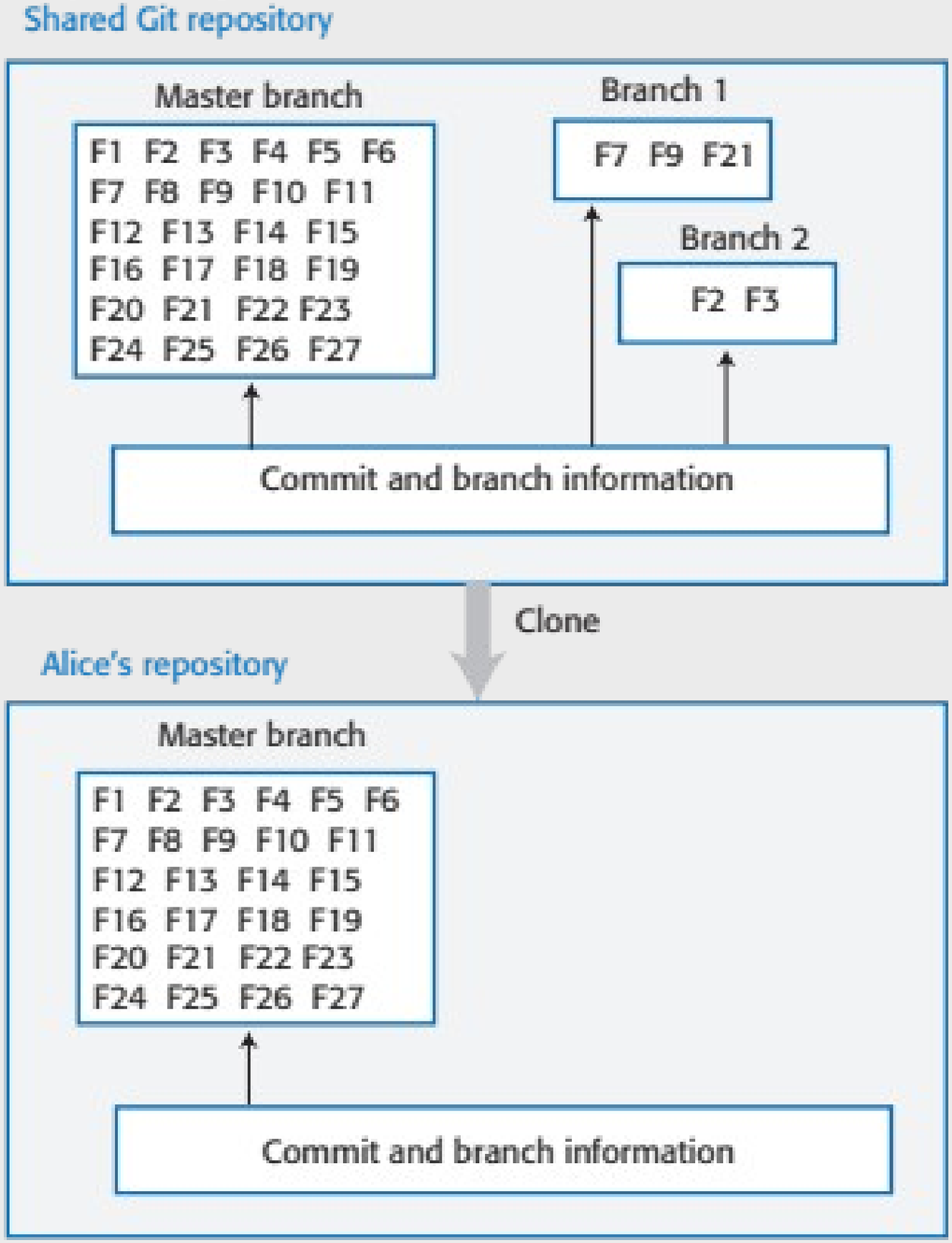
Git

- In 2005, Linus Torvalds, the developer of Linux, revolutionized source code management by developing a distributed version control system (DVCS) called Git to manage the code of the Linux kernel.

- This was geared to supporting large-scale open source development. It took advantage of the fact that storage costs had fallen to such an extent that most users did not have to be concerned with local storage management.

- Instead of only keeping the copies of the files that users are working on, Git maintains a clone of the repository on every user's computer

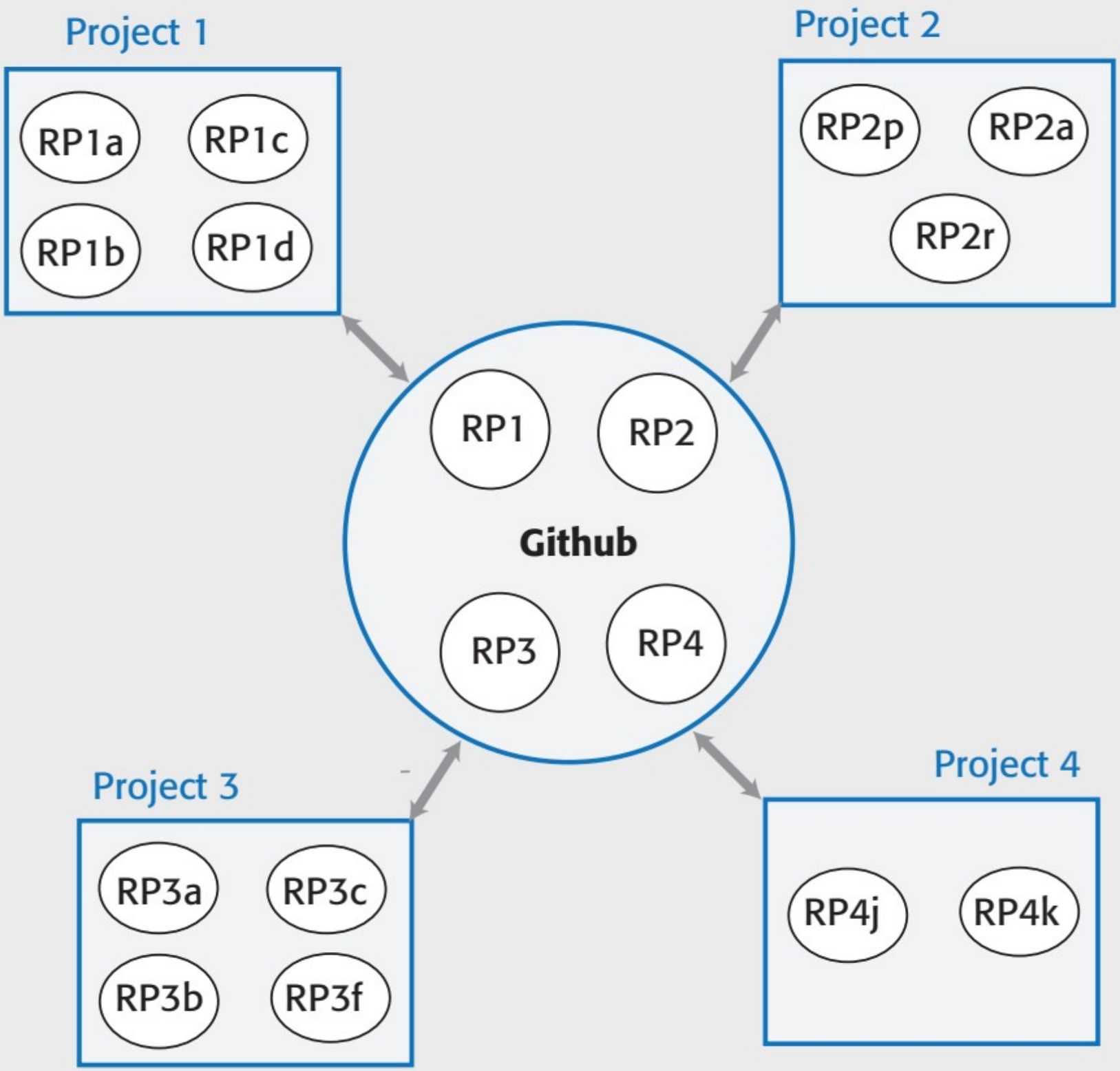
Figure 10.5 Repository cloning in Git



Benefits of distributed code management

- Resilience
  - Everyone working on a project has their own copy of the repository. If the shared repository is damaged or subjected to a cyberattack, work can continue, and the clones can be used to restore the shared repository. People can work offline if they don’t have a network connection.
- Speed
  - Committing changes to the repository is a fast, local operation and does not need data to be transferred over the network.
- Flexibility
  - Local experimentation is much simpler. Developers can safely experiment and try different approaches without exposing these to other project members. With a centralized system, this may only be possible by working outside the code management system.

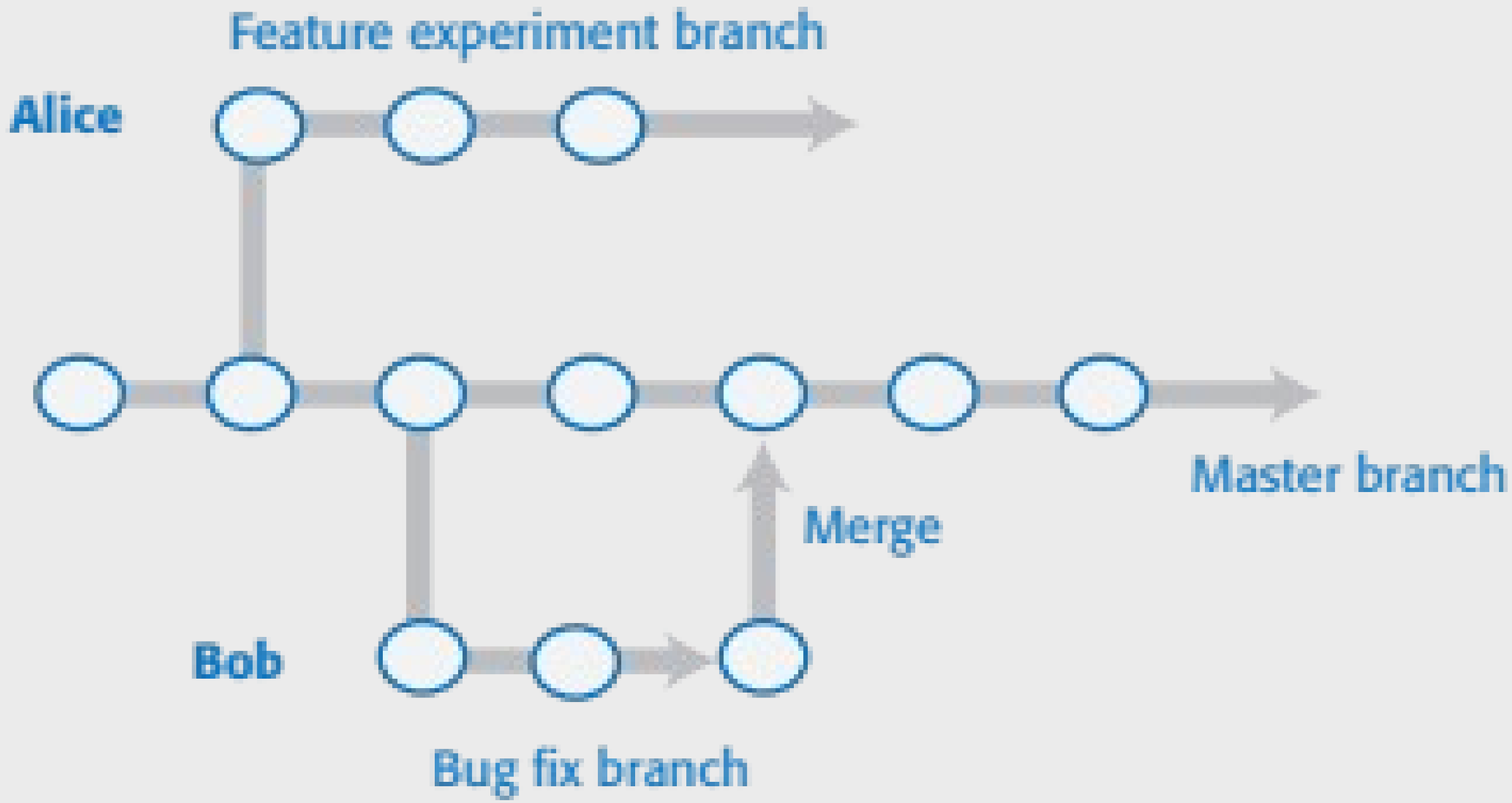
Figure 10.6 Git repositories



Branching and merging

- Branching and merging are fundamental ideas that are supported by all code management systems.
- A branch is an independent, stand-alone version that is created when a developer wishes to change a file.
- The changes made by developers in their own branches may be merged to create a new shared branch.
- The repository ensures that branch files that have been changed cannot overwrite repository files without a merge operation.
  - If Alice or Bob make mistakes on the branch they are working on, they can easily revert to the master file.
  - If they commit changes, while working, they can revert to earlier versions of the work they have done. When they have finished and tested their code, they can then replace the master file by merging the work they have done with the master branch

Figure 10.7 Branching and merging



DevOps automation

- By using DevOps with automated support, you can dramatically reduce the time and costs for integration, deployment and delivery.
- *Everything that can be, should be automated* is a fundamental principle of DevOps.
- As well as reducing the costs and time required for integration, deployment and delivery, process automation also makes these processes more reliable and reproducible.
- Automation information is encoded in scripts and system models that can be checked, reviewed, versioned and stored in the project repository.

Figure 10.5 Aspects of DevOps automation

Continuous integration
Each time a developer commits a change to the project's master branch, an executable version of the system is built and tested.

Continuous delivery
A simulation of the product's operating environment is created and the executable software version is tested.

Continuous deployment
A new release of the system is made available to users every time a change is made to the master branch of the software.

Infrastructure as code
Machine-readable models of the infrastructure (network, servers, routers, etc.) on which the product executes are used by configuration management tools to build the software's execution platform. The software to be installed, such as compilers and libraries and a DBMS, are included in the infastructure model.

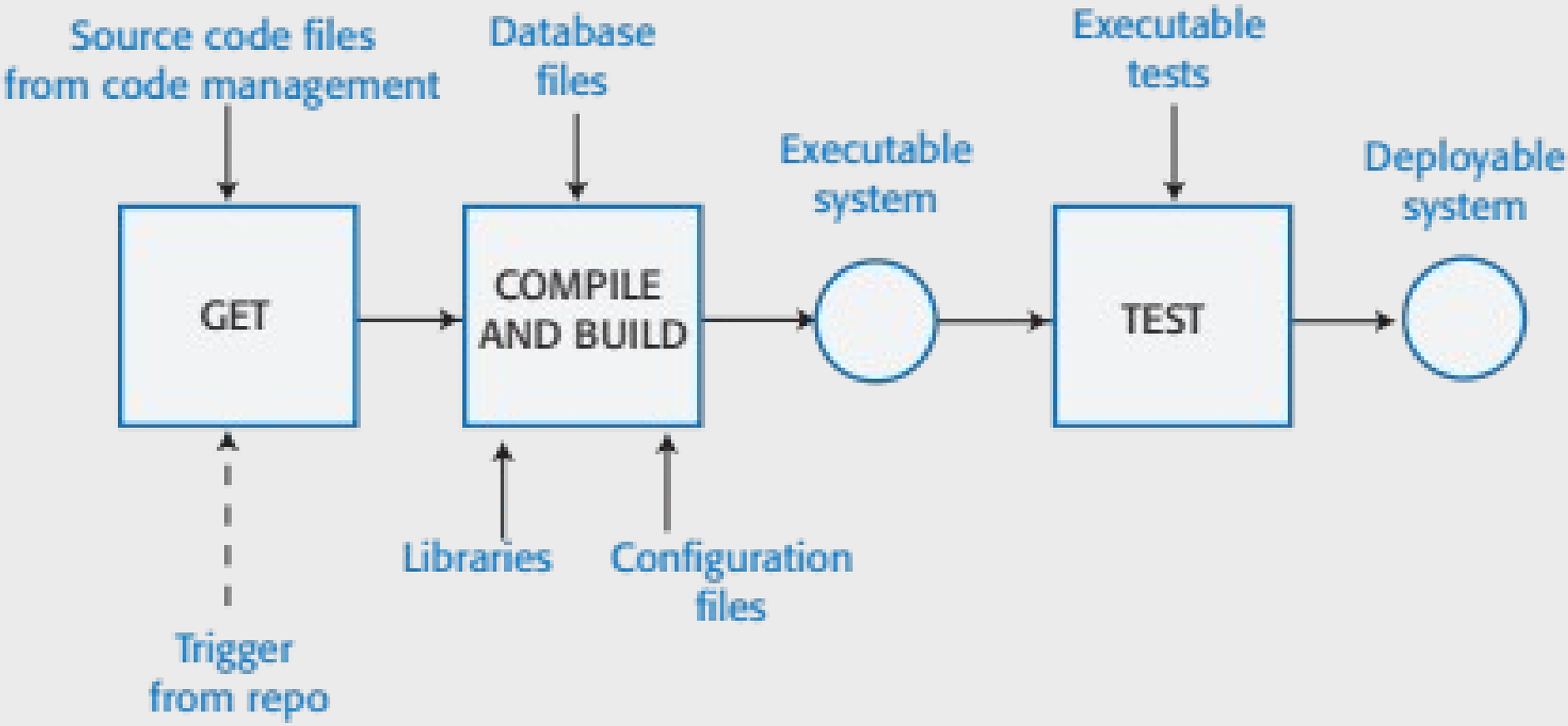
System integration

- System integration (system building) is the process of gathering all of the elements required in a working system, moving them into the right directories, and putting them together to create an operational system.
- Typical activities that are part of the system integration process include:
  - Installing database software and setting up the database with the appropriate schema.
  - Loading test data into the database.
  - Compiling the files that make up the product.
  - Linking the compiled code with the libraries and other components used.
  - Checking that external services used are operational.
  - Deleting old configuration files and moving configuration files to the correct locations.
  - Running a set of system tests to check that the integration has been successful.

Continuous integration

- Continuous integration simply means that an integrated version of the system is created and tested every time a change is pushed to the system's shared repository.
- On completion of the push operation, the repository sends a message to an integration server to build a new version of the product
- The advantage of continuous integration compared to less frequent integration is that it is faster to find and fix bugs in the system.
- If you make a small change and some system tests then fail, the problem almost certainly lies in the new code that you have pushed to the project repo.
- You can focus on this code to find the bug that's causing the problem.

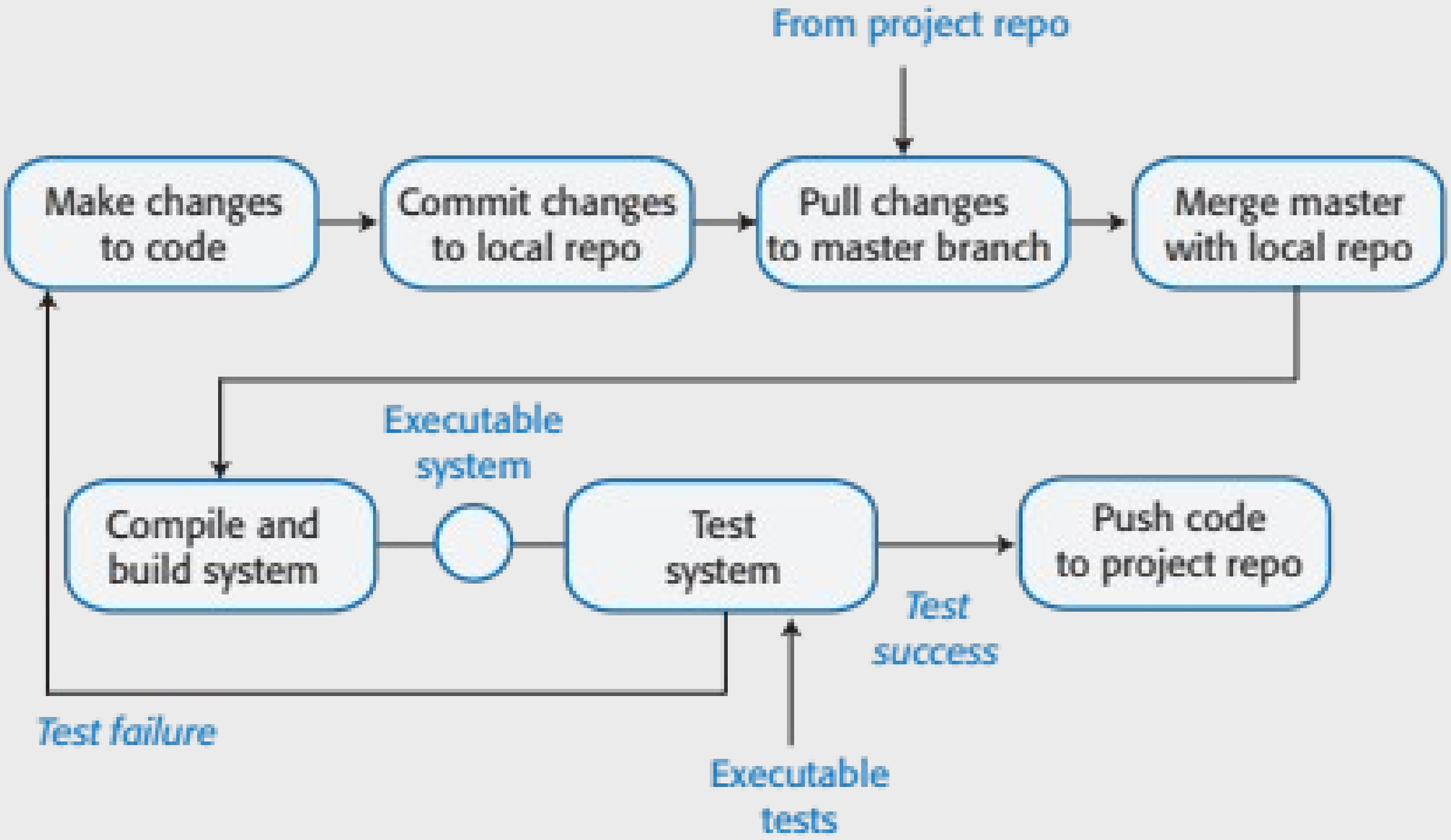
Figure 10.9 Continuous integration



Breaking the build

- In a continuous integration environment, developers have to make sure that they don't 'break the build'.
- Breaking the build means pushing code to the project repository which, when integrated, causes some of the system tests to fail.
- If this happens to you, your priority should be to discover and fix the problem so that normal development can continue.
- To avoid breaking the build, you should always adopt an 'integrate twice' approach to system integration.
  - You should integrate and test on your own computer before pushing code to the project repository to trigger the integration server

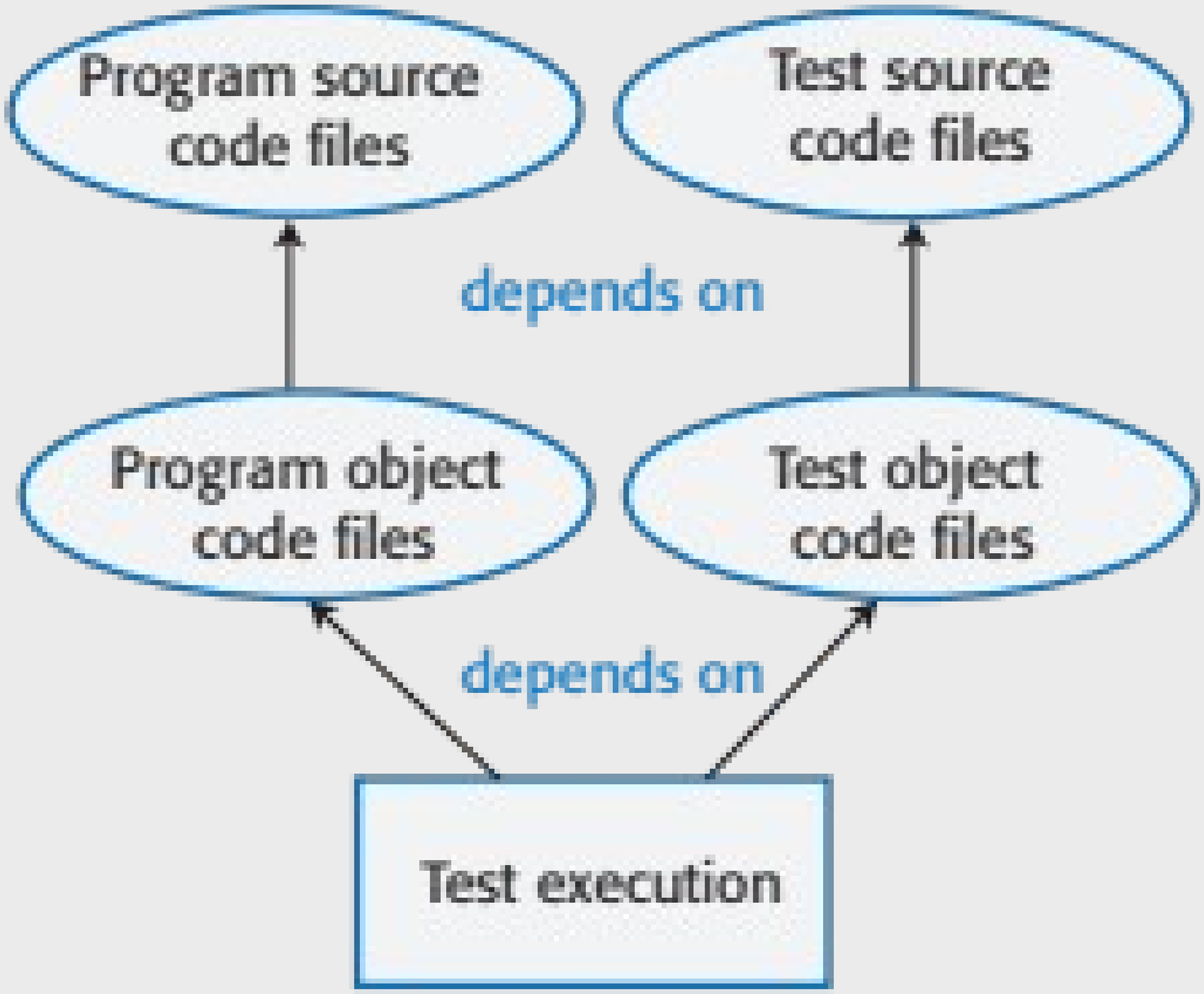
Figure 10.10 Local integration



System building

- Continuous integration is only effective if the integration process is fast and developers do not have to wait for the results of their tests of the integrated system.
- However, some activities in the build process, such as populating a database or compiling hundreds of system files, are inherently slow.
- It is therefore essential to have an automated build process that minimizes the time spent on these activities.
- Fast system building is achieved using a process of incremental building, where only those parts of the system that have been changed are rebuilt

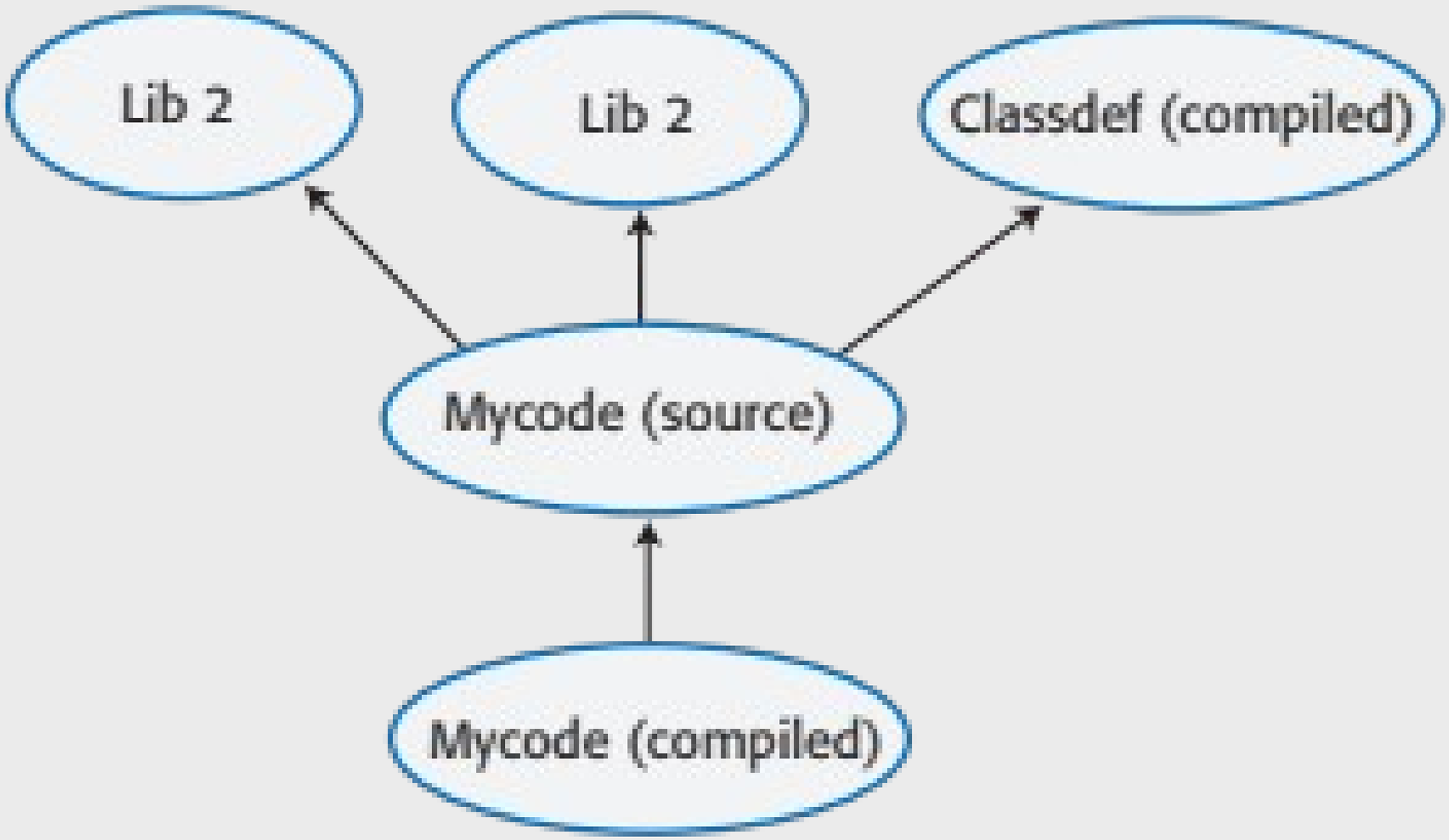
Figure 10.11 A dependency model



Dependencies

- Figure 10.11 is a dependency model that shows the dependencies for test execution.
- The upward-pointing arrow means ‘depends on’ and shows the information required to complete the task shown in the rectangle at the base of the model.
- Running a set of system tests depends on the existence of executable object code for both the program being tested and the system tests.
- In turn, these depend on the source code for the system and the tests that are compiled to create the object code.
- Figure 10.12 is a lower-level dependency model that shows the dependencies involved in creating the object code for a source code files called Mycode.

Figure 10.12 File dependencies

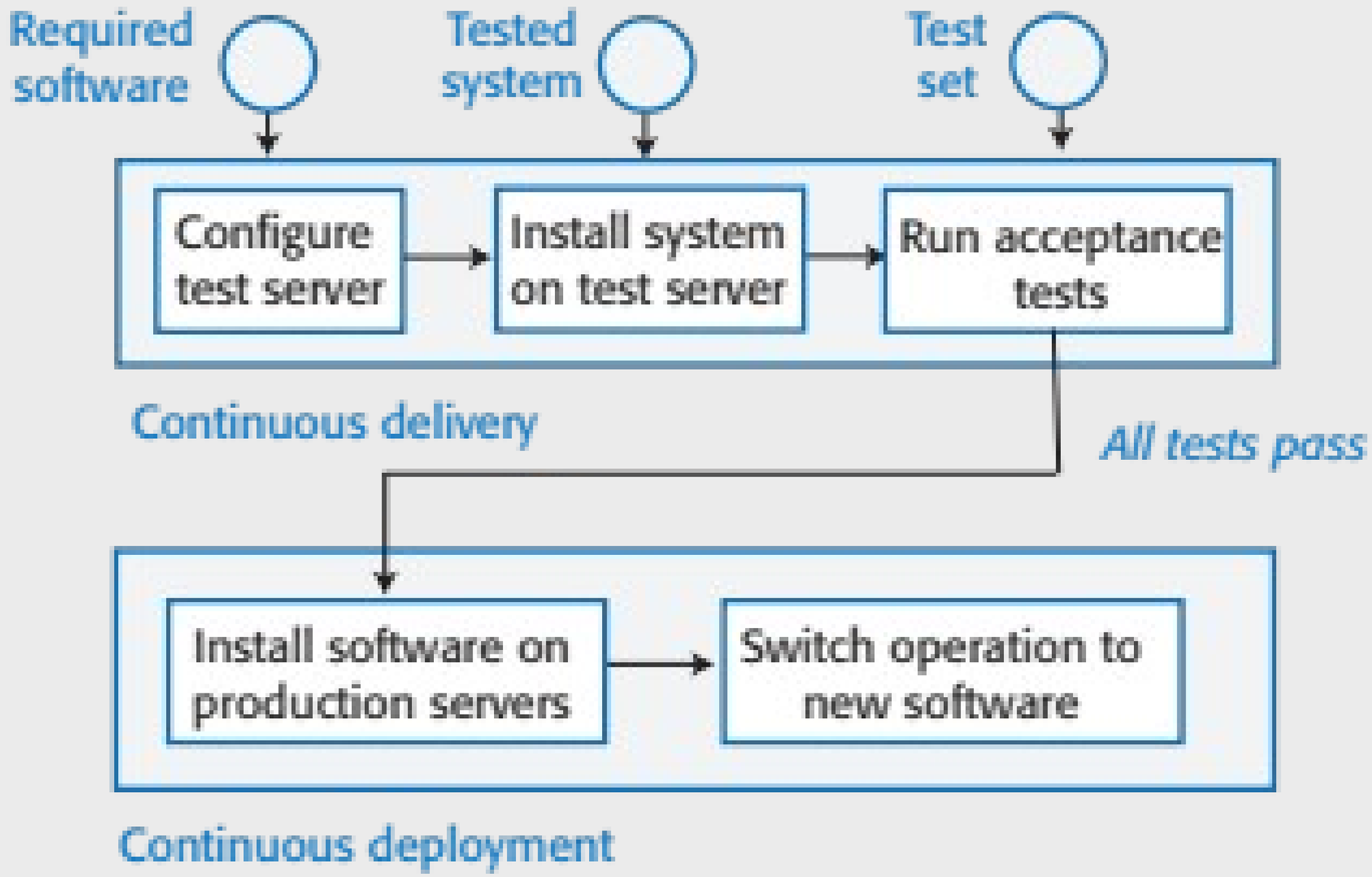


- An automated build system uses the specification of dependencies to work out what needs to be done. It uses the file modification timestamp to decide if a source code file has been changed.
  - The modification date of the compiled code is after the modification date of the source code. The build system infers that no changes have been made to the source code and does nothing.
  - The modification date of the compiled code is before the modification date of the compiled code. The build system recompiles the source and replaces the existing file of compiled code with an updated version.
  - The modification date of the compiled code is after the modification date of the source code. However, the modification date of Classdef is after the modification date of the source code of Mycode. Therefore, Mycode has to be recompiled to incorporate these changes.

Continuous delivery and deployment

- Continuous integration means creating an executable version of a software system whenever a change is made to the repository. The CI tool builds the system and runs tests on your development computer or project integration server.
- However, the real environment in which software runs will inevitably be different from your development system.
- When your software runs in its real, operational environment bugs may be revealed that did not show up in the test environment.
- Continuous delivery means that, after making changes to a system, you ensure that the changed system is ready for delivery to customers.
- This means that you have to test it in a production environment to make sure that environmental factors do not cause system failures or slow down its performance.

Figure 10.13 Continuous delivery and deployment



The deployment pipeline

- After initial integration testing, a staged test environment is created.
- This is a replica of the actual production environment in which the system will run.
- The system acceptance tests, which include functionality, load and performance tests, are then run to check that the software works as expected. If all of these tests pass, the changed software is installed on the production servers.
- To deploy the system, you then momentarily stop all new requests for service and leave the older version to process the outstanding transactions.
- Once these have been completed, you switch to the new version of the system and restart processing.

Figure 10.6 Benefits of continuous deployment

Reduced costs

If you use continuous deployment, you have no option but to invest in a completely automated deployment pipeline. Manual deployment is a time-consuming and error-prone process. Setting up an automated system is expensive and time-consuming but you can recover these costs quickly if you make regular updates to your product.

Faster problem solving

If a problem occurs, it will probably only affect a small part of the system and it will be obvious what the source of that problem is. If you bundle many changes into a single release, finding and fixing problems is more difficult.

Faster customer feedback

You can deploy new features when they are ready for customer use. You can ask them for feedback on these features and use this feedback to identify improvements that you need to make.

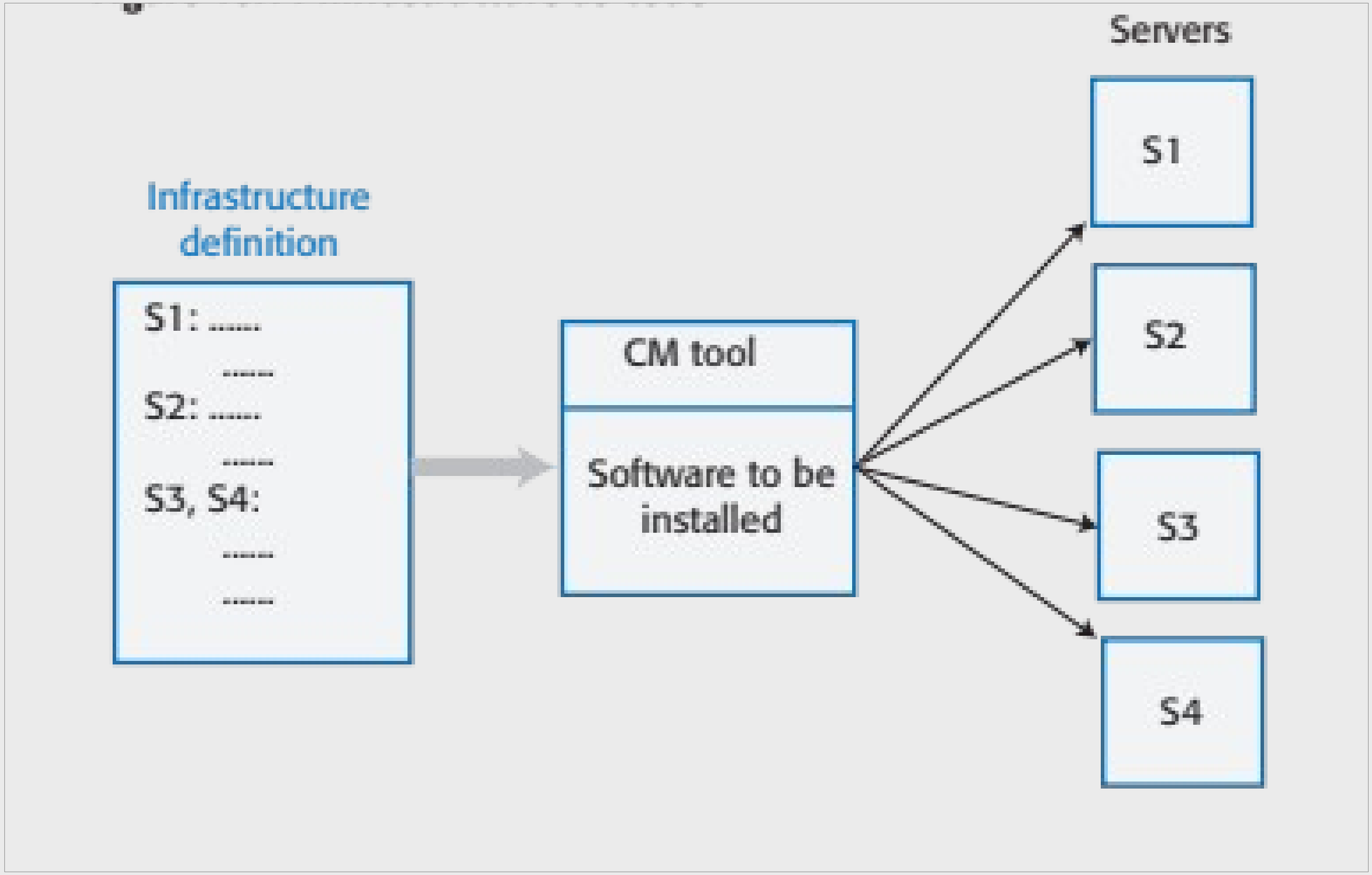
A/B testing

This is an option if you have a large customer base and use several servers for deployment. You can deploy a new version of the software on some servers and leave the older version running on others. You then use the load balancer to divert some customers to the new version while others use the older version. You can then measure and assess how new features are used to see if they do what you expect.

Infrastructure as code

- In an enterprise environment, there are usually many different physical or virtual servers (web servers, database servers, file servers, etc.) that do different things. These have different configurations and run different software packages.
- It is therefore difficult to keep track of the software installed on each machine.
- The idea of infrastructure as code was proposed as a way to address this problem. Rather than manually updating the software on a company's servers, the process can be automated using a model of the infrastructure written in a machine-processable language.
- Configuration management (CM) tools such as Puppet and Chef can automatically install software and services on servers according to the infrastructure definition

Figure 10.14 Infrastructure as code



Benefits of infrastructure as code

- Defining your infrastructure as code and using a configuration management system solves two key problems of continuous deployment.
  - Your testing environment must be exactly the same as your deployment environment. If you change the deployment environment, you have to mirror those changes in your testing environment.
  - When you change a service, you have to be able to roll that change out to all of your servers quickly and reliably. If there is a bug in your changed code that affects the system's reliability, you have to be able to seamlessly roll back to the older system.
- The business benefits of defining your infrastructure as code are lower costs of system management and lower risks of unexpected problems arising when infrastructure changes are implemented.

Table 10.7 Characteristics of infrastructure as code

Visibility
Your infrastructure is defined as a stand-alone model that can be read, discussed, understood and reviewed by the whole DevOps team.

Reproducability
Using a configuration management tool means that the installation tasks will always be run in the same sequence so that the same environment is always created. You are not reliant on people remembering the order that they need to do things.

Reliability
The complexity of managing a complex infrastructure means that system administrators often make simple mistakes, especially when the same changes have to be made to several servers. Automating the process avoids these mistakes.

Recovery
Like any other code, your infrastructure model can be versioned and stored in a code management system. If infrastructure changes cause problems you can easily revert to an older version and reinstall the environment that you know works.

Containers

- A container provides a stand-alone execution environment running on top of an operating system such as Linux.
- The software installed in a Docker container is specified using a Dockerfile, which is, essentially, a definition of your software infrastructure as code.
- You build an executable container image by processing the Dockerfile.
- Using containers makes it very simple to provide identical execution environments.
  - For each type of server that you use, you define the environment that you need and build an image for execution. You can run an application container as a test system or as an operational system; there is no distinction between them.
  - When you update your software, you rerun the image creation process to create a new image that includes the modified software. You can then start these images alongside the existing system and divert service requests to them.

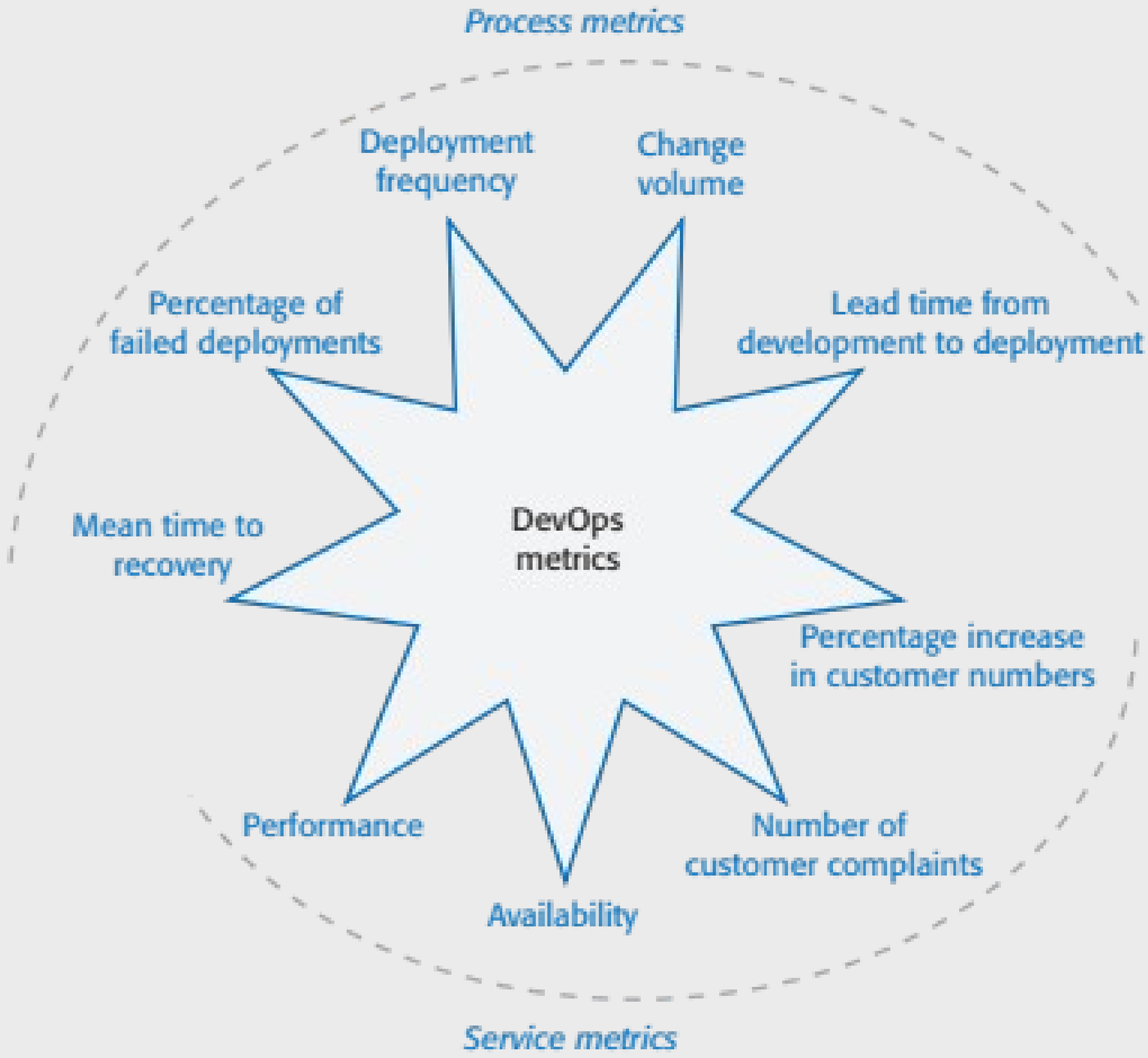
DevOps measurement

- After you have adopted DevOps, you should try to continuously improve your DevOps process to achieve faster deployment of better-quality software.
- There are four types of software development measurement:
  - **Process measurement** You collect and analyse data about your development, testing and deployment processes.
  - **Service measurement** You collect and analyse data about the software's performance, reliability and acceptability to customers.
  - **Usage measurement** You collect and analyse data about how customers use your product.
  - **Business success measurement** You collect and analyse data about how your product contributes to the overall success of the business.

Automating measurement

- As far as possible, the DevOps principle of automating everything should be applied to software measurement.
- You should instrument your software to collect data about itself and you should use a monitoring system, as I explained in Chapter 6, to collect data about your software's performance and availability.
- Some process measurements can also be automated.
- However, there are problems in process measurement because people are involved. They work in different ways, may record information differently and are affected by outside influences that affect the way they work.

Figure 10.15 Metrics used in the DevOps scorecard



Metrics scorecard

- Payal Chakravarty from IBM suggests a practical approach to DevOps measurement based around a metrics scorecard with 9 metrics:
  - These are relevant to software that is delivered as a cloud service. They include process metrics and service metrics
  - For the process metrics, you would like to see decreases in the number of failed deployments, the mean time to recovery after a service failure and the lead time from development to deployment.
  - You would hope to see increases in the deployment frequency and the number of lines of changed code that are shipped.
  - For the service metrics, availability and performance should be stable or improving, the number of customer complaints should be decreasing, and the number of new customers should be increasing.

Figure 10.16 Metrics trends

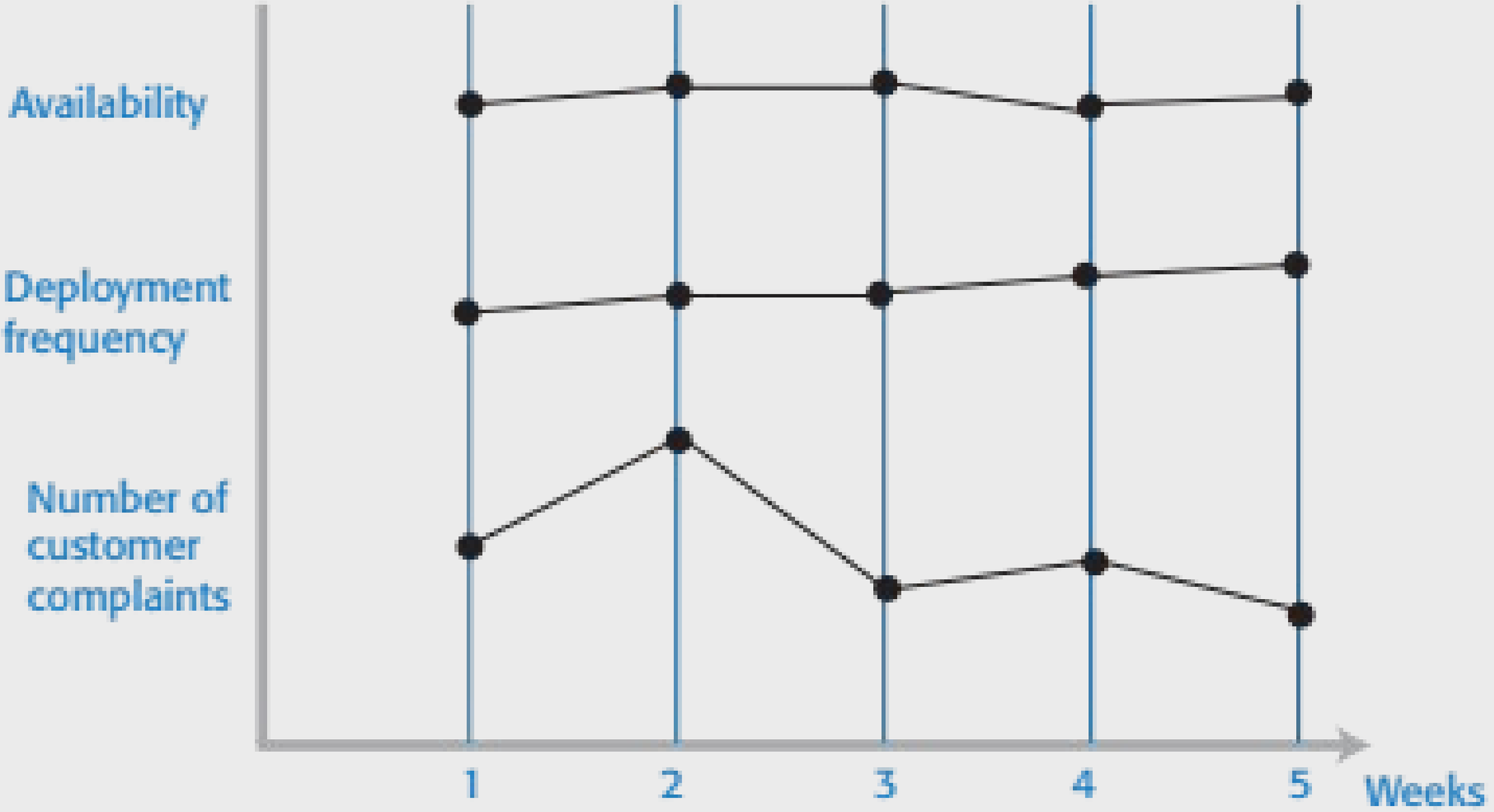
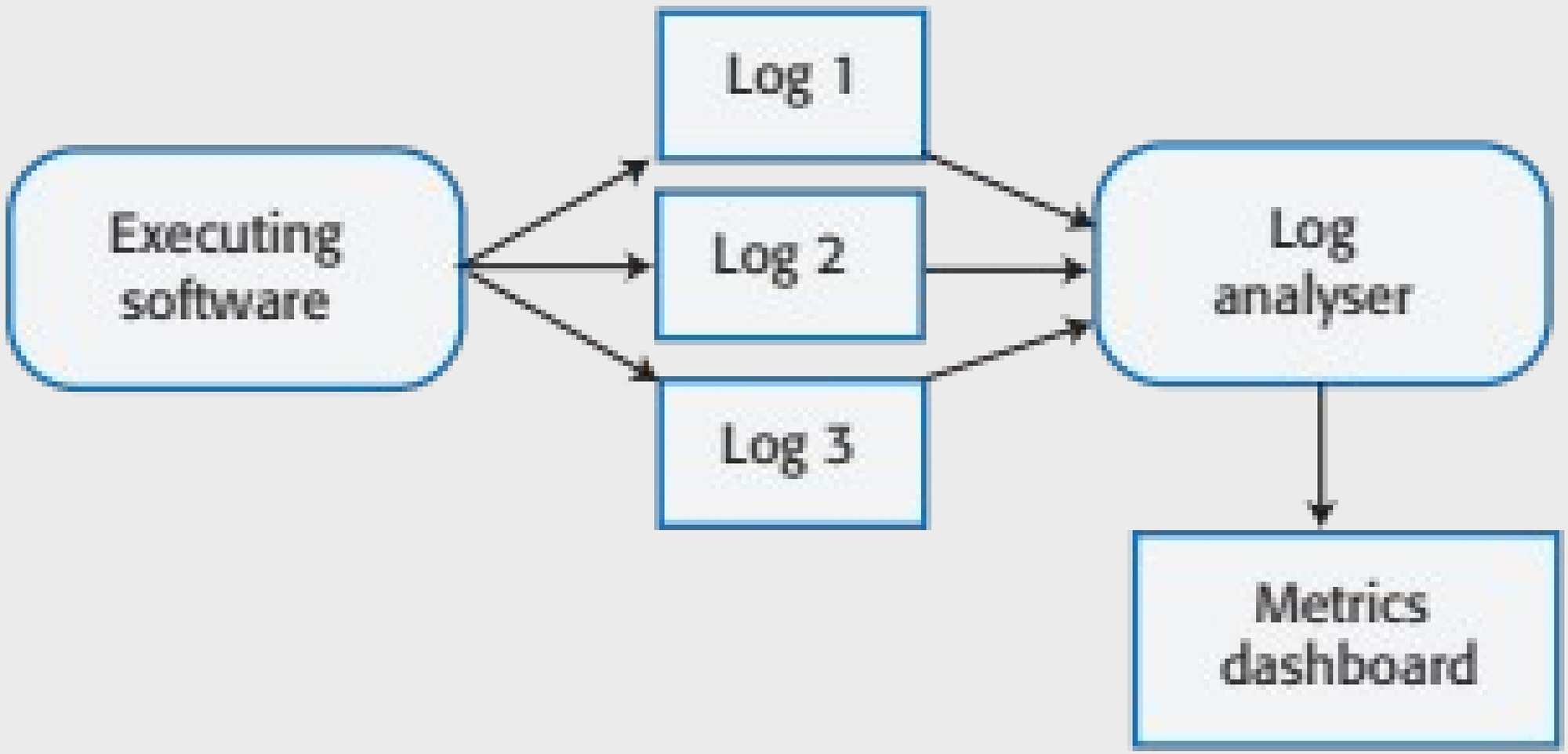


Figure 10.17 Logging and analysis



Key points 1

- DevOps is the integration of software development and the management of that software once it has been deployed for use. The same team is responsible for development, deployment and software support.
- The benefits of DevOps are faster deployment, reduced risk, faster repair of buggy code and more productive teams.
- Source code management is essential to avoid changes made by different developers interfering with each other.
- All code management systems are based around a shared code repository with a set of features that support code transfer, version storage and retrieval, branching and merging and maintaining version information.
- Git is a distributed code management system that is the most widely used system for software product development. Each developer works with their own copy of the repository which may be merged with the shared project repository.

Key points 2

- Continuous integration means that as soon as a change is committed to a project repository, it is integrated with existing code and a new version of the system is created for testing.
- Automated system building tools reduce the time needed to compile and integrate the system by only recompiling those components and their dependents that have changed.
- Continuous deployment means that as soon as a change is made, the deployed version of the system is automatically updated. This is only possible when the software product is delivered as a cloud-based service.
- Infrastructure as code means that the infrastructure (network, installed software, etc.) on which software executes is defined as a machine-readable model. Automated tools, such as Chef and Puppet, can provision servers based on the infrastructure model.
- Measurement is a fundamental principle of DevOps. You may make both process and product measurements. Important process metrics are deployment frequency, percentage of failed deployments, and mean time to recovery from failure.