Code Review & Reusability
✔ Best proof of reusability → If others can review, understand, and integrate the code easily.
✔ Test for reusability:
Give the code to someone else and see if they can merge and use it.
If they struggle, the code may need improvement.
✔ Key Purpose of Code Review:
Ensures that someone else understands and values the code before it's used.

Daniel Kahneman's "Thinking, Fast and Slow" & Debugging
✔ System 1: Fast, intuitive, high error rate → Used for quick decision-making.
✔ System 2: Slow, methodical, low error rate → Requires deliberate effort but is more accurate.
✔ Rubber Duck Debugging (Methodical Debugging Approach):
1. Explain the problem out loud
2. Go step by step through the code.
3. Think about each step clearly and logically.
4. Identify the issue that was previously overlooked.

Research on Code Reviews
✔ Code review catches 60-90% of errors (Fagan, 1976).
✔ The first reviewer matters the most (Cohen, 2006).
✔ Defect rates are related to program size, not necessarily other factors (El Imam, 2001).

Best Practices Before Sending Code for Review
✔ Make the code easy to review → Clean, structured, and well-documented.
✔ Keep changes small & focused → Easier to review and understand.
✔ Send early 'Work In Progress' (WIP) reviews → Get early feedback before finalizing.
✔ Review your own code first → Catch obvious mistakes before sending it for review.

PR/CL Descriptions (Pull Request/Change List)
When writing PR/CL descriptions, it's important to go beyond just describing what the change is. Instead, include the following:

✔ What: Briefly describe the change being made.
"Support @foo.com login."
✔ Why: Explain why the change is necessary or the reason behind it.
"Most users have foo.com login, and user research shows preference for passwordless login."
✔ How: Detail how the change accomplishes the goal and what was done to implement it.
"Give user options to enter password or click to login with foo, initiating OAuth flow in <design doc link>."
✔ Testing: Mention any testing conducted to verify the change.
"Added a test account test-login@foo.com."
"Added WebDriver tests that exercise the login."

Code Review Best Practices
✔ Flag Errors of Execution
During the code review, it's important to flag any execution errors such as:
Unclear documentation, Typos, Style violations, Bad or missing tests, Bugs
✔ Apply Deliberative Thinking
Ask questions to ensure the code is accurate and necessary:
- Is the algorithm correct?
- Is it built to specifications?
- Does this code need to exist?
- Is this the most elegant solution?

✔ Develop a Shared Understanding About the Purpose of the Code
- Align the team on "landmarks" and milestones for the project.
- Understand that small changes can lead to drifting from the original target.
- Each code review is an opportunity to course-correct the development process.
- Consider future use: "How will this code be used next year?"

✔ Establish N+1 Availability on Understanding of the Code
Make sure the team has a clear understanding of the code:
- Methods of code review should ensure everyone is on the same page.
- Projecting code in a meeting or conducting pair programming are effective strategies.
- Pull requests allow for code visibility and review from others, fostering shared understanding.

📌 Proper PR/CL descriptions and a thorough, deliberative code review process ensure that the code is understandable, accurate, and aligned with the project's goals, while allowing for necessary improvements and preventing errors.

Logging
1️⃣ Why Log? Logging is essential for various purposes in software development:
✔ Development: Helps in tracking application flow and ensuring everything works as expected during the development phase.
✔ Debugging: Assists in identifying issues and tracing errors by capturing specific details when something goes wrong.
✔ Security: Logs are used to track and detect unauthorized access or malicious activities in the system.
✔ Monitoring: Helps monitor system performance and health, allowing teams to react to issues proactively.
✔ Usage Insights: Logs can provide valuable insights into how users are interacting with the system and which features are being used the most.

2️⃣ How to Log?
Logging can be done in many ways, from simple print statements to using advanced logging libraries.
✔ Simple Logging (Print Statements): The most basic form of logging, but not scalable for large applications.
std::cout << "This is an info log." << std::endl;
✔ Using Logging Libraries:

Boost.Log: A more advanced and flexible logging library for C++ that provides logging with various severity levels.

```
#include <boost/log/trivial.hpp>
#include <iostream>

int main(int, char *[]) {
    BOOST_LOG_TRIVIAL(info) << "This is some info";       // Info log
    BOOST_LOG_TRIVIAL(warning) << "Not great...";        // Warning log
    BOOST_LOG_TRIVIAL(error) << "OH NOOOO";              // Error log
    BOOST_LOG_TRIVIAL(fatal) << "eff this I'm out";      // Fatal log
    return 0;
}
```

Logging Levels:

info: General information about system events.
warning: Events that are not critical but might need attention.
error: Critical issues that need fixing.
fatal: Severe issues that will cause the application to stop.

When to Log?

1️⃣ Application Startup and Shutdown
✔ Log when the application starts, including configuration details.
✔ Log when the application shuts down, including the reason (graceful exit or crash).

```
LOG_INFO("Application started successfully with config version 1.2.3");
LOG_INFO("Application shutting down due to SIGTERM signal");
```

2️⃣ Errors and Exceptions
✔ Log unexpected errors and exceptions with stack traces and relevant data.

```
try {
    int result = 10 / 0; // Division by zero
} catch (const std::exception& e) {
    LOG_ERROR("Exception caught: {}", e.what());
}
```

3️⃣ User Authentication and Authorization
✔ Log login attempts (success or failure) for security monitoring.
✔ Log permission-related issues for auditing purposes.

```
LOG_WARNING("Unauthorized access attempt by user: {}", username);
LOG_INFO("User {} logged in successfully", username);
```

4️⃣ Database Queries and Transactions
✔ Log slow queries to optimize performance.
✔ Log failed transactions to detect data inconsistencies.

```
LOG_INFO("Executing SQL query: SELECT * FROM users WHERE id=42");
LOG_ERROR("Database transaction failed: {}", errorMessage);
```

5️⃣ API Requests and Responses
✔ Log incoming API requests for debugging and analytics.
✔ Log response times to monitor performance bottlenecks.

```
LOG_INFO("Received API request: GET /users/42");
LOG_INFO("Response time: 120ms");
```

6️⃣ System Resource Usage
✔ Log system resource usage (memory, CPU, disk) periodically.
✔ Helps in detecting resource leaks or excessive consumption.

```
LOG_INFO("Current memory usage: 512MB, CPU usage: 45%");
```

7️⃣ Security-Related Events
✔ Log security incidents such as failed authentications, privilege escalation attempts, or suspicious activities.

```
LOG_CRITICAL("Multiple failed login attempts detected for user: {}", username);
```

What to log
✔ Correct Usage:
Log significant system events, such as errors, system behavior, or user interactions.
Log data that helps with debugging, performance monitoring, security, and usage insights.
✔ Wrong Information:
Log what is necessary and meaningful, but avoid unnecessary data that could compromise privacy or security.
✔ What NOT to Log:
Passwords
API keys, tokens, private encryption keys
Personally Identifiable Information (PII) (e.g., Social Security Numbers, credit card details)
Full, unfiltered user input (to prevent logging injection attacks)

📌Logging sensitive data like passwords can be a security risk. Instead, log a general error message without exposing any sensitive user details.

Where to Log
Logs should be structured to ensure they can be easily inspected, parsed, and persisted for future analysis.

✔ Goals for Logging:
Inspect manually → Allow developers to quickly find issues by reviewing logs.
Parse with tools → Logs should be in a structured format that tools can process automatically.
Do not fill the disk → Keep logs efficient in terms of size.
Persist after restart or crash → Ensure logs are not lost after system failures.
Resilient to hardware failures → Ensure logging continues despite temporary issues with the hardware.
✔ Log Sinks (Where Logs Are Stored):

1. Console (Standard Output / StdErr):
   Use for quick debugging during development or short-lived processes.
2. Log Files (Local Storage):
   Store structured logs locally for diagnostics and debugging.
   Logs can be saved locally before shipping to external log management tools.
3. Remote Server / Cloud-Based Log Management Services:
   Use when working at scale, such as with distributed systems, microservices, or large applications.
   These services provide search, filtering, visualization, and real-time monitoring.
   (AWS CloudWatch, Google Cloud Logging, Azure Monitor)

📌 Logs are stored in scalable and resilient systems that can be easily accessed and analyzed. Use appropriate log sinks depending on your application's scale and environment.

Log Levels
Log levels help categorize the severity and importance of the logged messages:
✔ Verbose/Debug:
Used for detailed, low-level information useful during development and debugging.
Example: "Variable x initialized to 5."
✔ Info:
Used for general information on the system's normal operation.
Example: "User logged in successfully."

✔ Warning:
Indicates a potential issue or unexpected event that isn't necessarily an error.
Example: "Disk space running low."
✔ Error:
Used when something goes wrong but doesn't stop the application.
Example: "Failed to connect to the database."
✔ Severe:
For critical issues that cause system failures or crashes.
Example: "Out of memory exception, application crashing."
✔ Considerations:

Info and higher should be logged to files for long-term storage.
Keep a copy of error prints to the console for quick visibility during debugging.

Logging to Disk
When logging at scale, you must prevent your disk from filling up:
✔ Log File Rotation:
Rotate log files to avoid filling up the disk.
Example:
When the log file exceeds a certain size (e.g., 10MB), rename the log file to include a timestamp (logname_YYYYMMDDhhmm.log).
Delete the oldest log files if you have 10 or more logs to cap storage usage at a limit (e.g., 100MB).

3️⃣ Logging vs. Privacy
Logging can involve sensitive data, so you must be mindful of privacy concerns:
✔ Real-World Example:
Google Street View Cars accidentally collected unencrypted WiFi data from 2006 to 2010, resulting in a $13 million settlement and significant damage to their reputation.
✔ Privacy Laws:
Companies must obtain consent, report data breaches, and allow data takeout under various regulations.GDPR (EU), CCPA (California), CMA (UK), DMA (EU)
✔ Challenges in Privacy Compliance:
Data deletion: Must delete data after a user deletes their account, but many logging systems are designed to be unmodifiable.
Tape backups: What about old backup tapes containing PII?

4️⃣ Privacy Solutions
These techniques are important to keep logs useful without compromising user privacy:
✔ Aggregation:
Aggregating data makes it non-personally identifiable.
Example: "How many total visits did we see last year?"
Aggregated logs can be safely kept indefinitely without violating privacy.
✔ Differential Privacy:
Helps prevent indirect inference of personal data from aggregated information.
Example: Using aggregated stats while ensuring individual data points can't be reconstructed.
✔ Pseudonymization:
Hashing personally identifiable information (PII) (e.g., visitor ID) and storing the hash instead of actual data.
HMAC-SHA2 hashes are hard to reverse (for now).
Example: Logging only the hashed version of a visitor's ID to count unique visitors.
Disclaimer: Pseudonymization may not be legally compliant depending on jurisdiction.

5️⃣ Privacy in Practice
✔ Privacy as a Primary Concern:
Privacy needs to be considered early in development and as a first-class concern.
Privacy and security go hand in hand, and data access controls are critical.
✔ Practical Implementation:
If you're building a product, ensure that it has a privacy section in your design documentation.
The product should undergo a privacy review before implementation and privacy approval after implementation before launch.

📌Logging is essential for system monitoring, but it must be done in a way that avoids privacy violations. Techniques like data aggregation, differential privacy, and pseudonymization help maintain compliance with privacy regulations while still providing useful logs for analysis. Always consider privacy and security together when building and deploying systems.