

COS30049 – Computing Technology Innovation Project

Week 11 – Building a Full-Stack Application (Debugging and Ensuring Code Quality)

(Lecture – 02)

Ningran Li (Icey)

ningranli@swin.edu.au



Acknowledgement of Country

We respectfully acknowledge the Wurundjeri People of the Kulin Nation, who are the Traditional Owners of the land on which Swinburne's Australian campuses are located in Melbourne's east and outer-east, and pay our respect to their Elders past, present and emerging.

We are honoured to recognise our connection to Wurundjeri Country, history, culture, and spirituality through these locations, and strive to ensure that we operate in a manner that respects and honours the Elders and Ancestors of these lands.

We also respectfully acknowledge Swinburne's Aboriginal and Torres Strait Islander staff, students, alumni, partners and visitors.

We also acknowledge and respect the Traditional Owners of lands across Australia, their Elders, Ancestors, cultures, and heritage, and recognise the continuing sovereignties of all Aboriginal and Torres Strait Islander Nations.



- Recap Front-end & Back-end
- React API Call
- Data Visualization
- Full-stack integration
- **Debugging and Troubleshooting**
- Best Practices in Code Development

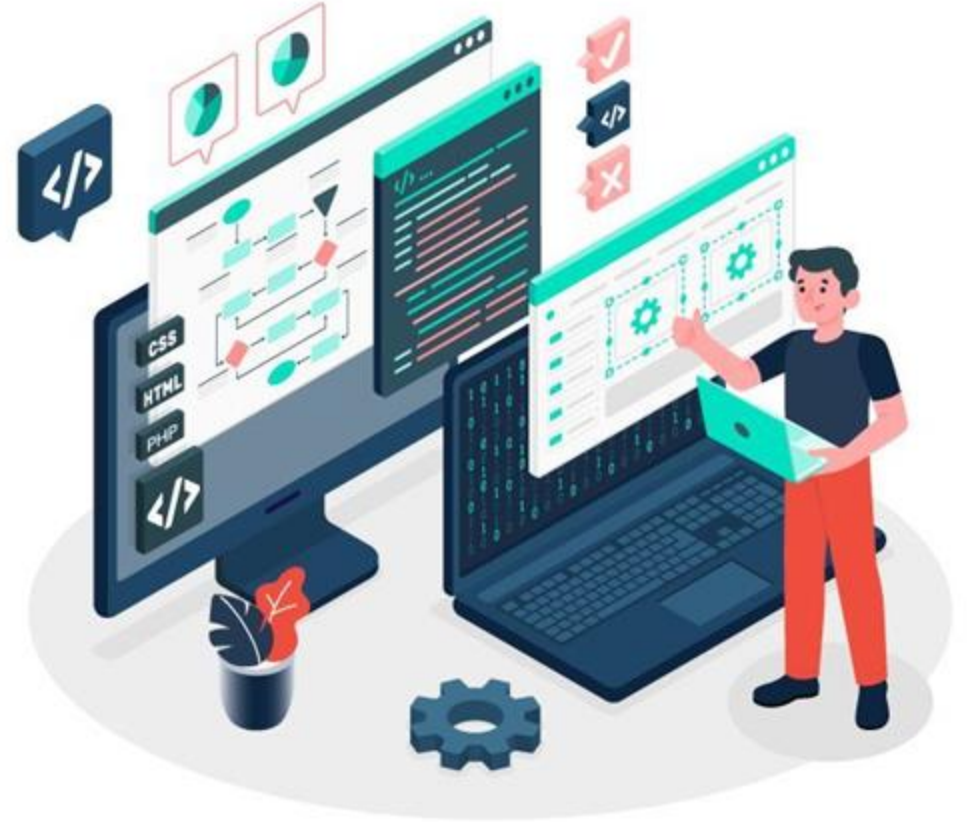


Image from [[Pinterest](#)]

Introduction to Debugging

What is Debugging?

- The process of identifying, analyzing, and removing errors (bugs) in software code
- Essential skill for developers to ensure software functions correctly

Why is Debugging Important?

- Improves software quality and reliability
- Saves time and resources in the long run
- Enhances understanding of the codebase

Types of Bugs

- Syntax Errors: Incorrect code structure
- Runtime Errors: Occur during program execution
- Logical Errors: Produce incorrect results without crashing

Debugging Process Overview

1. Reproduce the issue
2. Locate the source of the problem
3. Analyze the cause
4. Fix the bug
5. Test the solution

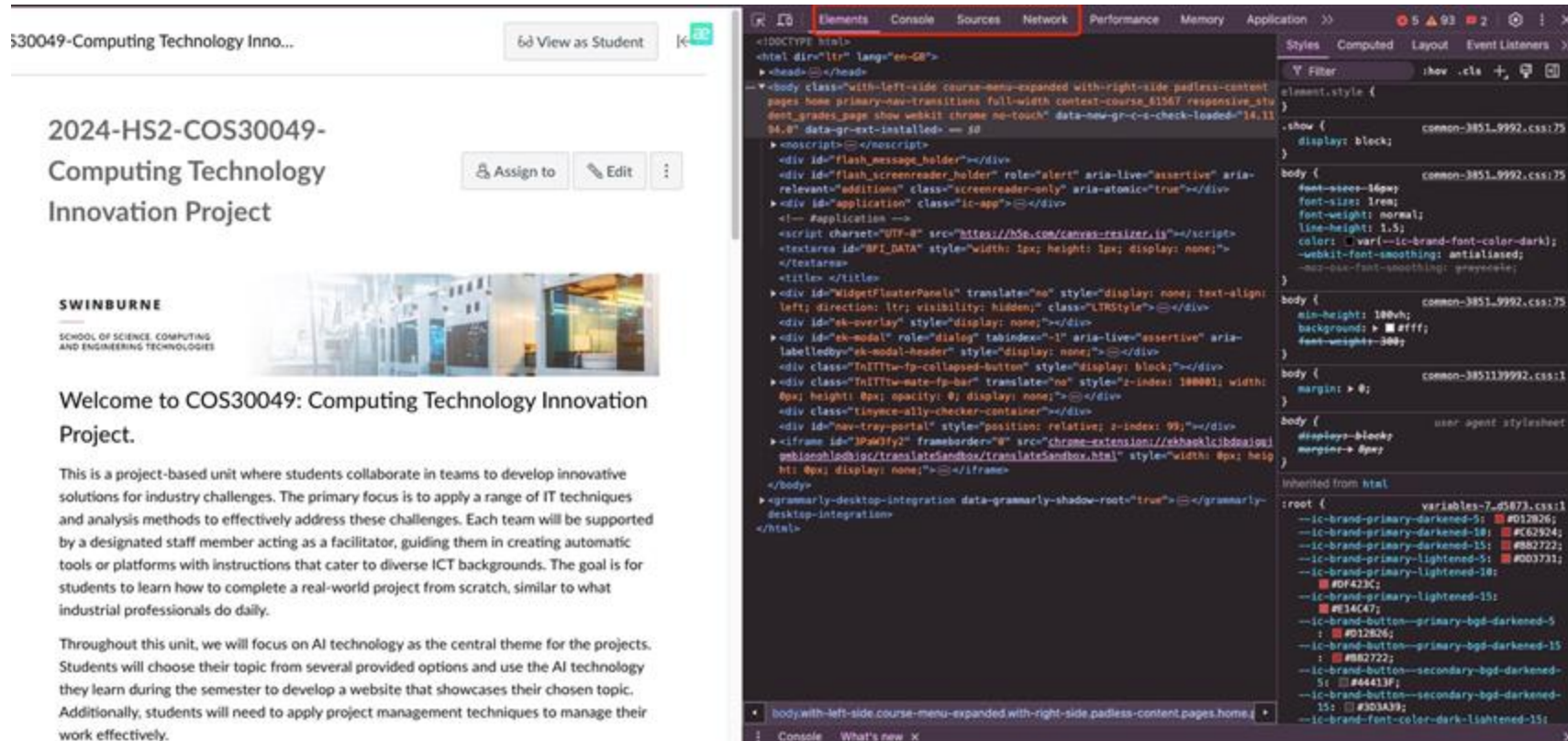


Image from [\[Pinterest\]](#)

Debugging Tools for React

Browser Developer Tools (F12)

- Elements tab: Inspect and modify DOM
- Console: View logs, errors, and warnings
- Sources: Set breakpoints and debug JavaScript
- Network: Monitor network requests and responses



Debugging Tools for React

React Developer Tools

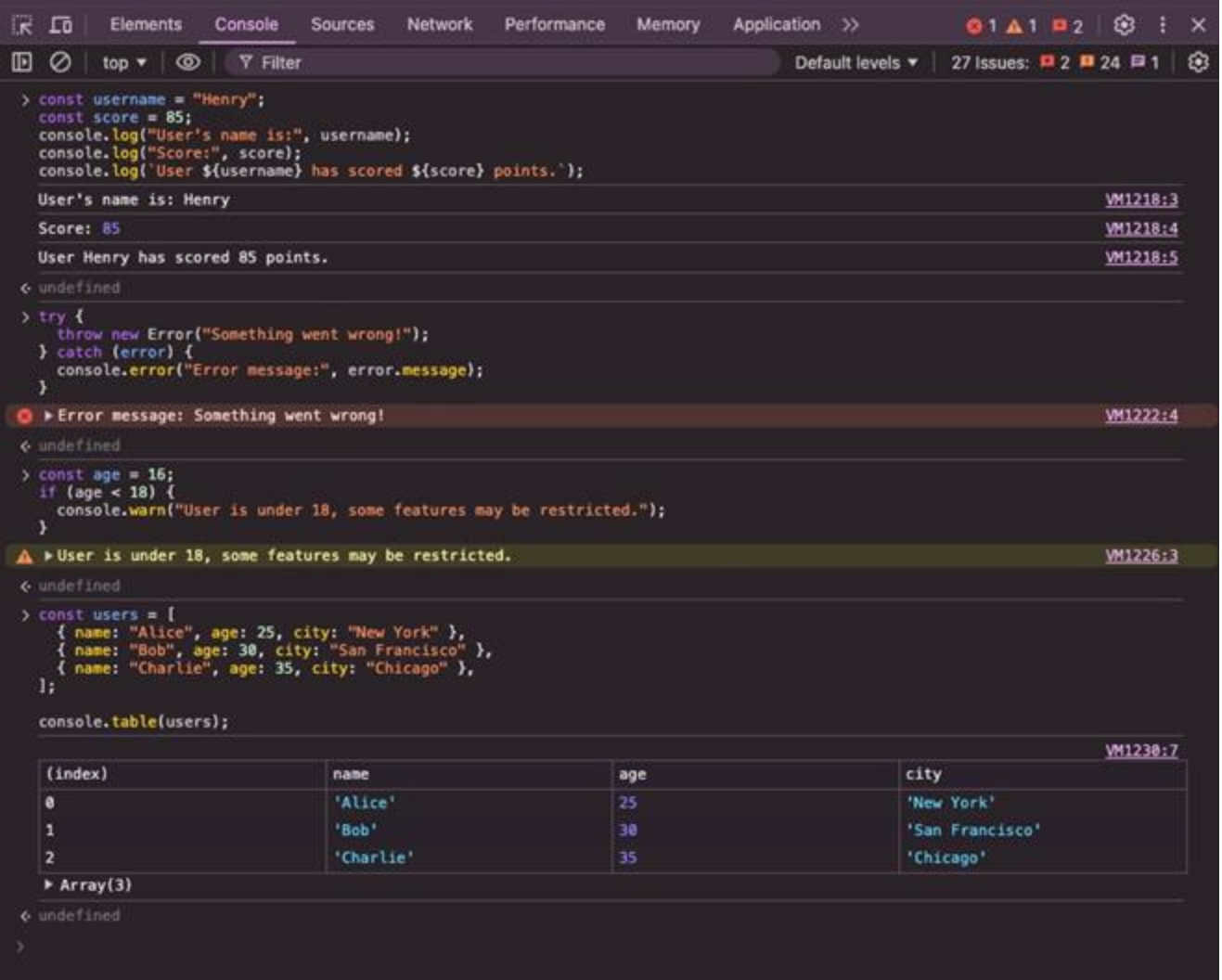
- Browser extension for debugging React applications
- Components tab: Inspect component hierarchy and props
- Profiler tab: Analyze component render performance

IDE Debugging Features

- Syntax Errors: Incorrect code structure
- Runtime Errors: Occur during program execution
- Logical Errors: Produce incorrect results without crashing

Console Logging

- `console.log()`: Output variable values and messages
- `console.error()`: Log error messages
- `console.warn()`: Log warning messages
- `console.table()`: Display tabular data



The screenshot shows a web browser's developer console with the following content:

```
> const username = "Henry";
const score = 85;
console.log("User's name is:", username);
console.log("Score:", score);
console.log(`User ${username} has scored ${score} points.`);

User's name is: Henry
Score: 85
User Henry has scored 85 points.

< undefined

> try {
  throw new Error("Something went wrong!");
} catch (error) {
  console.error("Error message:", error.message);
}

Error message: Something went wrong!

< undefined

> const age = 16;
if (age < 18) {
  console.warn("User is under 18, some features may be restricted.");
}

User is under 18, some features may be restricted.

< undefined

> const users = [
  { name: "Alice", age: 25, city: "New York" },
  { name: "Bob", age: 30, city: "San Francisco" },
  { name: "Charlie", age: 35, city: "Chicago" },
];

console.table(users);
```

(index)	name	age	city
0	'Alice'	25	'New York'
1	'Bob'	30	'San Francisco'
2	'Charlie'	35	'Chicago'

Array(3)

Debugging Techniques for React

Using React Error Boundaries

- Catch and handle errors in component trees
- Prevent entire app from crashing due to component errors

Debugging State and Props

- Use React Developer Tools to inspect component state and props
- Implement logging in lifecycle methods or hooks

Performance Profiling

- Use React Profiler to identify performance bottlenecks
- Analyze component render times and optimize as needed

```
1 class ErrorBoundary extends React.Component {
2   constructor(props) {
3     super(props);
4     this.state = { hasError: false };
5   }
6
7   static getDerivedStateFromError(error) {
8     return { hasError: true };
9   }
10
11  componentDidCatch(error, errorInfo) {
12    console.log('Error caught:', error, errorInfo);
13  }
14
15  render() {
16    if (this.state.hasError) {
17      return <h1>Something went wrong.</h1>;
18    }
19    return this.props.children;
20  }
21 }
```

Debugging Tools for FastAPI

Uvicorn Debug Mode

- Run FastAPI with `--reload` flag for automatic reloading
- Example: `uvicorn main:app --reload --port 8000`

FastAPI's Built-in Documentation

- Access interactive API documentation at `/docs` endpoint
- Test API endpoints directly from the browser

Python Debugger (pdb)

- Built-in Python debugging tool
- Set breakpoints, step through code, inspect variables

Logging in FastAPI

- Use Python's built-in `logging` module
- Configure log levels and formats

```
1 # Python Debugger
2 import pdb
3
4 @app.get("/debug-example")
5 async def debug_example():
6     pdb.set_trace() # Debugger will pause here
7     result = complex_calculation()
8     return {"result": result}
9
10
11 # Logging in FastAPI
12 import logging
13
14 logging.basicConfig(level=logging.DEBUG)
15 logger = logging.getLogger(__name__)
16
17 @app.get("/log-example")
18 async def log_example():
19     logger.debug("This is a debug message")
20     logger.info("This is an info message")
21     logger.warning("This is a warning message")
22     return {"message": "Check the logs"}
```


Debugging Techniques for FastAPI

```
1 # Exception Handling
2 # Example of a global exception handler
3 from fastapi import FastAPI, Request
4 from fastapi.responses import JSONResponse
5 app = FastAPI()
6 @app.exception_handler(Exception)
7 async def global_exception_handler(
8     request: Request, exc: Exception):
9     return JSONResponse(
10         status_code=500,
11         content={"message": f"An unexpected \
12 error occurred: {str(exc)}"}
13 )
14
15 # Request Validation Debugging
16 from pydantic import BaseModel, validator
17 class User(BaseModel):
18     username: str
19     email: str
20     age: int
21
22     @validator('age')
23     def check_age(cls, v):
24         if v < 18:
25             raise ValueError('Must be 18 or older')
26         return v
27
28 # Middleware for Debugging
29 import time
30 @app.middleware("http")
31 async def add_process_time_header(request: Request, call_next):
32     start_time = time.time()
33     response = await call_next(request)
34     process_time = time.time() - start_time
35     response.headers["X-Process-Time"] = str(process_time)
36     return response
```

Exception Handling

- Use try-except blocks to catch and handle exceptions
- - Implement global exception handlers for consistent error responses

Request Validation Debugging

- Use Pydantic models for request body validation
- - Implement custom validators for complex validation logic

Middleware for Debugging

- Implement custom middleware for logging or debugging
- - Track request/response cycles, execution time, etc.

Full Stack Debugging Strategies

End-to-End Testing

- Implement integration tests that cover both frontend and backend
- Use tools like Cypress or Selenium for automated E2E testing

API Testing

- Use tools like Postman or Insomnia for API testing
- Create and run automated API tests

Network Debugging

- Use browser Network tab to inspect API calls
- Monitor request/response cycles, headers, and payloads

Cross-Origin Resource Sharing (CORS) Debugging

- Understand CORS errors and how to resolve them



Image from [[Eficode](#)]

Tools for Full Stack Development

Version Control Systems

- Git: Distributed version control system
- GitHub/GitLab: Platforms for hosting and collaborating on Git repositories



Image from [\[medium.com\]](https://medium.com)

Continuous Integration/Continuous Deployment (CI/CD) Tools

- Jenkins: Open-source automation server
- GitLab CI: Integrated CI/CD platform
- GitHub Actions: CI/CD platform integrated with GitHub



Image from [\[learntek\]](https://learntek.com)

Monitoring and Logging Tools

- ELK Stack (Elasticsearch, Logstash, Kibana): For log management and analysis
- Prometheus and Grafana: For metrics collection and visualization



Image from [\[medium.com\]](https://medium.com)

Container and Orchestration Tools

- Docker: Platform for containerizing applications
- Kubernetes: Container orchestration platform for scaling and managing containerized applications

- Recap Front-end & Back-end
- React API Call
- Data Visualization
- Full-stack integration
- Debugging and Troubleshooting
- **Best Practices in Code Development**



Image from [[Pinterest](#)]

Code Development Best Practices

Why are following Best Practices Important?

- Improve code quality and readability
- Enhance maintainability and scalability
- Reduce bugs and technical debt
- Facilitate team collaboration and knowledge sharing

Benefits of High-Quality Code

- Faster development cycles
- Easier onboarding for new team members
- Reduced long-term costs
- Improved software reliability and performance



Image from [\[Pinterest\]](#)

Code Style and Consistency

Importance of Coding Standards

- Ensures uniform code appearance across the project
- Reduces cognitive load when reading and reviewing code
- Facilitates easier maintenance and updates

Popular Style Guides

- Python: PEP 8 (Python Enhancement Proposal 8)
- JavaScript/React: Airbnb React/JSX Style Guide
- General: Google Style Guides for various languages

Enforcing Code Style

- Use linters and formatters:
 - Python: flake8, black
 - JavaScript: ESLint, Prettier
- Integrate with CI/CD pipeline for automated checks

```
1 # Example: Python Code Style (PEP 8)
2
3 # Good
4 def calculate_average(numbers):
5     """Calculate the average of a list of numbers."""
6     if not numbers:
7         return 0
8     return sum(numbers) / len(numbers)
9
10 # Bad
11 def calculateAverage( numbers ):
12     if not numbers: return 0
13     return sum(numbers)/len(numbers)
```

Clear Code Structure

Importance of Clear Code Structure

- A **clear code structure** ensures **consistency**, making code easier to read, navigate, and **maintain**. It **supports collaboration** by helping team members understand the project quickly. Organized code simplifies debugging and allows for **smoother scaling** as the project grows. In short, it improves efficiency and reduces errors in development.

Good Code Structure Guides for React

A typical React project should follow a well-organized structure that separates concerns and promotes reusability:

```
my-react-app/  
├── public/  
│   └── index.html           # Main HTML file  
├── src/  
│   ├── assets/             # Static assets like images, fonts, etc.  
│   ├── components/         # Reusable React components  
│   ├── services/           # API calls or business logic  
│   ├── hooks/              # Custom React hooks  
│   ├── pages/              # Different pages or views  
│   ├── styles/             # Global CSS or SASS files  
│   ├── App.js              # Main App component  
│   ├── index.js            # Entry point of the application  
│   └── App.css              # App-level styling  
├── .env                    # Environment variables  
├── package.json            # Project dependencies and scripts  
└── README.md               # Documentation
```

Clear Code Structure

Good Code Structure Guides For FastAPI (python)

A well-structured FastAPI project encourages modularity and scalability. Here's an ideal layout for a FastAPI project:

```
my_fastapi_project/
├── app/
│   ├── api/
│   │   ├── v1/
│   │   │   ├── endpoints/
│   │   │   └── __init__.py
│   │   └── __init__.py
│   ├── core/
│   ├── models/
│   ├── services/
│   ├── db/
│   ├── utils/
│   ├── main.py
│   ├── __init__.py
│   └── tests/
├── .env
├── requirements.txt
└── README.md
```

API routes

API endpoints (e.g., auth, users, items)

Application settings, configuration, and security

Pydantic models *for* data validation or ORM models

Business logic, background jobs, external API calls

Database models and session management

Utility functions

Entry point to start FastAPI app

Unit and integration tests

Environment variables

Python dependencies

Documentation

Version Control Best Practices

Git Workflow

1. Create a feature branch from main/master
2. Make small, focused commits
3. Pull and rebase regularly to stay up-to-date
4. Create a pull request for code review
5. Merge after approval and passing CI checks

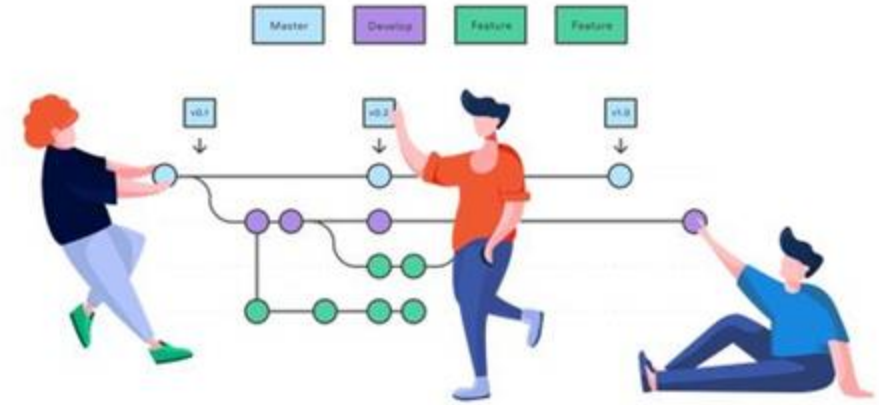


Image from [[Pinterest](#)]

Meaningful Commit Messages

- Use the imperative mood: "Add feature" instead of "Added feature"
- Provide context in the commit body
- Reference issue numbers if applicable
- Example:

```
```\nfeat: Add user authentication system\n- Implement JWT-based authentication\n- Create login and registration endpoints\n- Add middleware for protected routes\nCloses #issue123\n```
```

# Code Comments and Documentation

## When and How to Write Comments

- Explain "why" rather than "what" the code does
- Use comments for complex algorithms or business logic
- Keep comments up-to-date with code changes

## Types of Documentation

### 1. Inline Documentation

Docstrings for functions, classes, and modules

### 2. README Files

Project overview, setup instructions, and usage examples

Badges for build status, code coverage, etc.

### 3. API Documentation

Use tools like Swagger/OpenAPI for RESTful APIs

Provide examples and expected responses

### 4. Architecture and Design Documents

High-level system design and component interactions

Data flow diagrams and entity-relationship diagrams

```
1 # Inline Documentation
2
3 def calculate_discount(price: float, percentage: float) -> float:
4 """
5 Calculate the discounted price.
6
7 Args:
8 price (float): The original price.
9 percentage (float): The discount percentage (0-100).
10
11 Returns:
12 float: The price after applying the discount.
13
14 Raises:
15 ValueError: If percentage is not between 0 and 100.
16 """
17 if not 0 <= percentage <= 100:
18 raise ValueError("Percentage must be between 0 and 100")
19 return price * (1 - percentage / 100)
```

# Code Complexity Management

## Keeping Functions and Methods Simple

- Follow the Single Responsibility Principle (SRP)
- Aim for functions under 20-30 lines of code
- Use descriptive names to self-document code

## Avoiding Code Duplication (DRY Principle)

- Extract common functionality into reusable functions or classes
- Use inheritance and composition effectively in object-oriented programming
- Implement utility functions for frequently used operations

```
1 # Example of simplifying complex code
2
3 # Before
4 def process_data(data):
5 result = []
6 for item in data:
7 if item['status'] == 'active':
8 if item['type'] == 'user':
9 result.append({
10 'id': item['id'],
11 'name': item['name'],
12 'email': item['email']
13 })
14 elif item['type'] == 'admin':
15 result.append({
16 'id': item['id'],
17 'name': item['name'],
18 'role': 'administrator'
19 })
20 return result
```

```
22 # After
23 def is_active(item):
24 return item['status'] == 'active'
25
26 def process_user(item):
27 return {
28 'id': item['id'],
29 'name': item['name'],
30 'email': item['email']
31 }
32
33 def process_admin(item):
34 return {
35 'id': item['id'],
36 'name': item['name'],
37 'role': 'administrator'
38 }
39
40 def process_data(data):
41 processors = {
42 'user': process_user,
43 'admin': process_admin
44 }
45 return [processors[item['type']](item)
46 for item in data if is_active(item)]
```

# Code

## Importance of Code Review

- Improves overall code quality
- Catches bugs and security issues early
- Shares knowledge across the team
- Ensures adherence to coding standards and best practices

## Effective Code Review Techniques

- Use a checklist to ensure consistency
- Focus on the architecture and design first, then details
- Provide constructive feedback with explanations
- Use automated tools to catch style issues before human review
- Keep reviews small and frequent (under 400 lines of code per review)



Image from [\[pinterest\]](#)



# Code Review

## Code Review Checklist

- Does the code follow the project's style guide?
- Is the code well-documented and easy to understand?
- Are there any potential security vulnerabilities?
- Is the code efficient and performant?
- Are edge cases handled properly?
- Are there appropriate unit tests?

Example Code Review Comment:

```

In the `process_payment` function:

Consider using a decimal type instead of `float` for currency calculations to avoid precision issues. Also, it might be good to add a check for negative amounts to prevent invalid payments.

Suggested change:

```
from decimal import Decimal
```

```
def process_payment(amount: Decimal):  
    if amount <= Decimal('0'):  
        raise ValueError("Payment amount must be positive")  
    # Rest of the function...  
```
```

# Code Refactoring

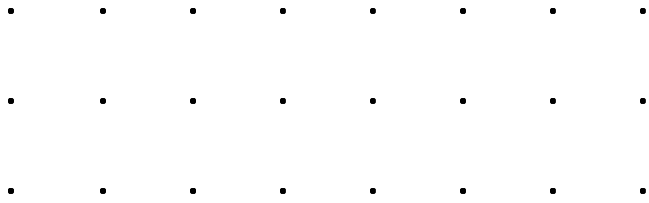
## Identifying Refactoring Opportunities

- Code smells (e.g., duplicate code, long methods, large classes)
- Declining performance
- Difficulty in adding new features
- Increasing technical debt

## Common Refactoring Techniques

- Extract Method: Break down long methods into smaller, focused ones
- Rename: Improve naming of variables, methods, and classes
- Move Method: Relocate methods to more appropriate classes
- Replace Conditional with Polymorphism: Use polymorphism instead of complex conditionals
- Introduce Parameter Object: Group related parameters into a single object

```
1 // Example of refactoring (JavaScript)
2 // Before
3 function calculateTotal(items) {
4 let total = 0;
5 for (let i = 0; i < items.length; i++) {
6 total += items[i].price * items[i].quantity;
7 if (items[i].type === 'food') {
8 total *= 0.9; // 10% discount on food items
9 }
10 }
11 return total;
12 }
13
14 // After refactoring
15 function calculateTotal(items) {
16 return items.reduce((total, item) => total
17 + calculateItemTotal(item), 0);
18 }
19
20 function calculateItemTotal(item) {
21 const baseTotal = item.price * item.quantity;
22 return applyDiscount(baseTotal, item.type);
23 }
24
25 function applyDiscount(total, itemType) {
26 return itemType === 'food' ? total * 0.9 : total;
27 }
```



**Thank you**

