Quality Software

This report documents my understanding of quality software developed throughout the Fall 2025 semester. Quality software became tangible when reviewing code or working with poorly designed systems during project development.

Throughout this semester's work, security emerged as a critical consideration. Input validation and authentication protecting sensitive operations became essential priorities when building projects, where improper validation could expose vulnerabilities.

Simplicity proved vital for maintainability during iterative development. High complexity made systems difficult to extend during later project phases. Simple implementations enabled more effective iteration and future team collaboration across codebases.

Documentation's value became clear when returning to earlier work or enabling others to use projects. Systems that new team members could run without external assistance demonstrated quality documentation, reducing friction and enabling efficient collaboration.

This Documentation

Throughout this documentation I use the REST API PHP Server Project as an example along with other callouts from previous frameworks and docs I've written. Example Documentation Published to Medium. This Graduate MCP smart hub research project.

- **Documentation.md** - Documentation standards and practices
- **Frameworks.md** - Power of frameworks with examples
- **Principles.md** - General principles from experience
- **Testing.md** - Testing principles and approaches
- **Report.md** - Weekly activities from semester work

General Principles

Some general principles I follow up

Fail Fast

It's important to not over analyze everything and get to writing code. I do believe architecting solutions is a good pre requisite, but am also cautious of over architecting.

Often times when architecting there is a gap between the high level overview and the actual implementation limitations that are learned during development.

Don't Repeat Yourself

This one is pretty cliche these days, but I think its a useful thing to keep in mind when writing software.

It's important to reuse functionality. This way it can be extended and maintained in one place.

Separation of Concerns

Microservicing can be a good approach but there is definitely a spectrum. Over microservicing has its own problems with maintenance of many separate services along with their deployments. While massive

monoliths can be difficult to maintain for reasons in the inverse.

Authentication scattered across 14 endpoint files means updating auth touches 14 files. Authentication in middleware means one place, one change.

If changing the database requires touching API route handlers, your concerns aren't separated. Good separation means swapping the database only touches the database layer.

Finding the right balance whether microservices vs monolith or code organization depends on team size and complexity. Start simple and split when you need to.

Environment Isolation

Dev, staging, and prod should be as similar as possible but completely isolated.

All of the environmental differences we have should be configurable. So we have ymls or the equivalent that contain the configs we need for separate environments.

In the case of cloud environments we can store the configurations in the cloud environment and pull from there during run/build time.

Automate Repetitive Tasks

If it becomes a repetitive task then we should automate it, like homework 4.

Manual process:

- SSH into server
- Pull latest code
- Run migrations
- Restart service
- Check logs

Scripted:

```
./deploy.sh staging
```

This way we can kick off the script whenever we make changes or deploy, and can mostly be assured that the behaviour is the same.

Iterate and optimize over time

This is a big one for me. Don't try to solve every problem at the outset or over optimize.

First we need to actually develop and deploy an MVP like application. From there we can iterate and optimize based on user feedback and metrics.

It's important to collect good metrics like API calls, CPU usage, etc. From there we have a benchmark to compare iterations against.

Keep It Simple

Probably one of the most important to me.

High complexity becomes so difficult to maintain and iterate on. It also becomes difficult for other developers to understand and maintain later.

Developing low complexity applications can actually be difficult. I've found this especially true with AI. AI so often wants to implement the most complex solution possible. Understanding code and what you are doing helps immensely when guiding AI solutions.

The Power of Frameworks

Frameworks solve common problems so we don't have to. Security, validation, routing, and auth are common features - frameworks provide well tested implementations so we can focus on business logic/use cases.

Key benefits: Speed to production, security by default, community support, less code to maintain.

Trade-offs: Learning curve, abstraction complexity, dependency on maintainers, potential bloat. Learning curve is definitely a consideration, some frameworks are like an extension of an existing programming language.

Frameworks are great for enterprise applications that need security, validation, auth, etc.

FastAPI - Python Web Framework

FastAPI is ubiquitous in enterprises using Python for APIs. Teams often switch to Python specifically to use FastAPI.

```python
from fastapi import FastAPI
from pydantic import BaseModel

app = FastAPI()

class Agent(BaseModel):
    name: str
    status: str

@app.post("/agents")
async def create_agent(agent: Agent):
    return {"id": 1, "name": agent.name, "status": agent.status}
```

What you get:

- Pydantic validation gives clients a clear input/output contract
- OpenAPI docs at `/docs` - interactive API explorer (no separate documentation needed)
- Type hints enable IDE autocomplete
- Async support for concurrent requests
- Clear validation errors as JSON

OpenAI Python SDK - Client Library Design

This python SDK is awesome for LLM api applications.

```python
from openai import OpenAI
```

```
client = OpenAI(api_key=api_key)
response = client.chat.completions.create(
    model="gpt-5",
    ...
)
```

Handles automatically:

- Authentication, headers, API key management
- Retries with exponential backoff
- Rate limiting and quota tracking
- Type-safe responses
- Streaming support

This framework is everywhere and can be used with many models outside of openai's own.

FastMCP - MCP Server Framework

FastMCP handles MCP protocol details, is becoming the standard along with the low level MCP SDK by anthropic.

```
from fastmcp import FastMCP

mcp = FastMCP("Database Tools")

# decorator for tool creation
@mcp.tool()
def search_records(query: str, limit: int = 10) -> str:
    """Search database records by query string"""
    return f"Found results for: {query}"
```

Framework handles: Protocol implementation, tool registration, serialization, validation, error handling.

This framework is awesome for standing up and building an mcp server.

Why Frameworks are Important

Security: Auth, SQL injection protection, secure password hashing. Easier than writing a custom solution in many cases, or we can extend the framework for our use case.

Developer Experience: Clear patterns, excellent docs, active communities, IDE support, generated API documentation.

Constant Iteration: This is especially true for popular open source frameworks. The communities are comprised of many developers that contribute updates and features constantly.

Well Tested: Frameworks are often well tested and have been used for many production use cases.

Maintenance: Community security patches, bug fixes, performance optimizations, backward compatibility.

Velocity: Ship features faster. Less boilerplate means more time on unique business logic.

Real-World Impact

FastAPI's `/docs` endpoint saves hours. Developers test APIs interactively without separate documentation or curl commands.

OpenAI SDK handles rate limiting automatically. Implementing exponential backoff, quota tracking, and retry logic yourself takes days and still misses edge cases.

Frameworks are force multipliers. They handle solved problems correctly so you spend time on what makes your project unique.

They give us the scaffolding we can build and iterate on for future development.

Testing and Quality Assurance

Why Testing Matters

Untested code is likely to contain issues

Manual testing locally is definitely a good start and can catch issues.

Automated tests catch edge cases and errors after code changes.

Integration tests are great and ensure that the entire app works end-to-end as expected. These can be integrated into your CICD pipelines to ensure apps function as expected prior to deploying.

Cost of Bugs

- Find it while coding: Fix it immediately, minimal cost
- Find it during QA: Someone files a ticket, you context-switch back, big cost
- Find it in production: Users affected, emergency fix, reputation hit, max cost

> *Testing can prevent bugs from reaching produciton. Although it won't catch every edge case or environmental difference, it provides a good line of defense against bugs be introduced during initial deployment or further iterations.*

Testing Approaches in my HW4

Automated API Testing

Built `test_api.sh` that hits all 14 endpoints. Tests auth, validates parameters, checks error handling.

Runs the entire suite in seconds. Catches regressions immediately after changes. Also doubles as API documentation through examples.

Swagger docs for apis are awesome for local dev and even higher envs. It provides an easy interface for developers and other users to leverage when interacting/testing the endpoints for APIs.

Browser-Based Testing

Created `test.html` with forms for each endpoint and live response display.

Good for visual confirmation and testing the actual user experience.

A lot of things can be automated, but in my experience UI testing is still very much a manual visual check by developers. Checking the UI prior to pushing/deploying changes still(for now at least) needs a human touch.

Database Testing

Directly queried the database to verify foreign keys work, cascading deletes happen correctly, data integrity holds.

Database testing can be difficult, especially in enterprise systems. A huge reason(especially with cloud) is the differences in compute/storage across envs. It can be hard to replicate PROD usage in lower envs.

The infra cost of completely replicating PROD in lower environments can also be hard to justify. So iteration and usage metrics can be useful for estimations, but often we won't get it perfect on the first iteration.

All of that being said, ensuring data integrity, data quality, and testing apps integration is important. Ensuring that when the application is running in PROD that it isn't encountering unforeseen table locking, read/write issues is important, and can be tested for in the lower envs.

Testing Implementation Timeline

Week 2: Building the Test Suite

Following Milestone 2 from the project plan, I completed the testing infrastructure:

Stage 1 REST API Testing Suite (Complete):

- Testing suite: cURL script (test_api.sh) + browser suite (test.html)
- NGINX deployment with automated start/stop scripts and self-contained config
- All 14 REST API endpoints (7 public GET, 7 protected with bearer tokens)
- PHP + MySQL backend with PDO, bearer token authentication
- 5 database tables (agents, tasks, tools, logs, api_tokens)

This testing suite referenced throughout the sections above wasn't hypothetical - I built it in Week 2 and used it continuously throughout development. Building comprehensive tests early meant every subsequent code change was validated immediately, catching regressions before they became problems.

The test_api.sh script became both a testing tool and API documentation through examples, demonstrating the dual value of well-designed tests.

What Good Tests Cover

Happy path - normal usage with valid data produces expected results

Error cases - invalid inputs return useful error messages instead of crashing

Edge cases - empty strings, null values, max lengths, special characters all handled

Security - auth actually prevents unauthorized access, SQL injection attempts fail safely

Breaking Changes - arguably the most important to me. Breaking changes being caught before being deployed to production is a big saver of time and prod issues.

Documentation

Documentation Basics

Something super close to my own heart is documentation. This was especially critical pre AI, maybe not so much now. Although I still run into a good number of gotchas.

What makes documentation useful is a good title so other devs can easily find it when searching for relevant docs for their use case.

So something like 'API Setup' probably isn't as useful as 'API Gateway Infrastructure setup with Authentication'. The latter title gives far more context and is specific to AWS APIs with the callout of Authentication.

Step by step and Screenshots are the biggest. The best documentation is written while walking through the implementation step by step with plenty of screenshots along the way.

Here is an example of something I've published to be publically available (can't show internal enterprise docs of course)

Example Documentation Published to Medium

Documentation as Code

Code should be self documenting is a common cliche that often gets repeated.

This is valid for certain languages and less so for others. So for example, python is super easy to read and for the most part, understand what is going on. Java, however, is not. It can be difficult to discern what is happening in Java because of the layers of abstractions being used.

Anyone using AI (which is most folks these days) knows that most models include comments in the code they write. I often find these comments too verbose for a lot of these cases and they can be really noisy.

Comments in code are awesome, but they should be used when something isn't clear or could be confusing. Especially with multithreaded py functions, it is good to have a solid doc string to say what this function does and what it interacts with. Making use of keywords to catch a devs eye is important. So for python we have NOTE, HACK, etc. Those keywords can be great as callouts for reminders, fixes, etc.

Also including github issue links in code I've found can be super useful. Even for me with my own code it is good to see an issue that outlines why I made a dev decision because of a library's limitation or design.

What I look for in Docs

In my own experience when looking at docs to solve a problem I have or looking for information there are a few of things I'm watching for.

I want the docs to get right to the point. I dont want a huge generic preamble that tells me nothing.

I want clear examples with code. So many times folks who write documentation do not include code or only snippets. At least link to the source code, it is so frustrating when a snippet is missing context for how it is being used or doesn't work in isolation because of dependency issues.

Plenty of screenshots, this is the biggest. Step by steps with screenshots can be quickly skimmed and used as a reference.

How I do it

I've written some really popular internal documentation in a few companies. This is great for my own reference as well as for other devs, and gives me visibility outside my immediate team/org.

When I've struggled developing something of my own that had many gotchas, internal limitations, etc. I'll put together some thorough documentation.

This can also be great for other team members. If you ever leave or switch teams then your docs live on and can be maintained in your absence.

I follow the wants I listed above. I make sure to include Step by Step instructions, pre requisites, plenty of screenshots, and a reference table with links.

The best time to write docs is right after the implementation I find. So after its fresh in my memory I'll walk through each piece, take screenshots, and include each step.

Being too verbose is FAR BETTER than skipping over steps and leaving the person using your docs wondering how you got from A -> B.

Documentation Evolution This Semester

Week 2: Initial Documentation

Documentation Created (Complete):

- Marp presentation with 30+ slides, endpoint documentation, and screenshots
- PDF export with embedded images
- README with quick-start guide, .env setup, NGINX/PHP usage, and testing instructions
- Code backup created (code-backup.zip)

Starting with comprehensive documentation from the beginning meant the project was accessible to others immediately, not as an afterthought.

Week 5: Living Documentation Practice

With the major project milestone complete, focus shifted to finalizing the Marp documentation for CSC 640. This documentation captures the quality software principles demonstrated throughout the semester and serves as a comprehensive guide for building maintainable, scalable systems.

Updated report files to reflect current project status and weekly progress. Quality documentation is a living artifact that evolves alongside the project - keeping it current ensures it remains valuable for team members and stakeholders.

Week 6: Refinement and Voice

Updated and refined quality software documentation for HMW5. Rewrote introduction, principles, frameworks, testing, and documentation files with authentic voice and practical examples. Replaced textbook SOLID principles with real-world principles from experience. Switched from Laravel-specific examples to modern frameworks (FastAPI, OpenAI SDK, FastMCP). Added enterprise database testing insights. Condensed README and removed redundant content. Documentation now reflects professional experience rather than generic CS theory.

Reflection: This documentation itself demonstrates the principles I outlined above. Started comprehensive (Week 2), maintained regularly (Week 5), refined iteratively (Week 6). Documentation is never "done" - it evolves with understanding and remains a living guide for current and future team members.