DevOps – Project Rapport

Group e - Soufflé

May 23, 2023

Course: DevOps
Course ID: BSDSESM1KU
Course manager: Helge Pfeiffer

Group members:

Name:	**E-mail:**
Asger Clement Nebelong Lysdahl	asly@itu.dk
Laurits Munk Kure	laku@itu.dk
Patrick Bohn Mathiassen	pmat@itu.dk
Rasmus Overgaard Olesen	raoo@itu.dk

Contents

1 System's Perspective

1.1 MiniTwit Application

We chose to rewrite MiniTwit to C# to utilize Blazor WebAssembly, which allows us to write all code in the same language (Microsoft 2023b; Microsoft 2023a). Blazor additionally runs as a single-page application which is faster at loading, partly because data can be fetched in the background and because full-page reloads are rare. One downside of WebAssembly is the long initial load of the application, which hopefully can be reduced or removed with future versions of DotNet. Our application depends on some NuGet packages to communicate with the rest of the system (NuGet 2023).

Identity Server	An easy and secure way of managing users (Software 2023).
Prometheus-net	Sets up the /metrics endpoints for Prometheus.
Serilog.Sinks.ElasticSearch	Gives Serilog a log sink for ElasticSearch, letting it send logs directly to ElasticSearch
EF Core	O/RM for communicating with the database

We aimed for our system to have a simple and maintainable architecture, by utilizing as few different tools as possible and by striving to follow some architectural patterns. These were Clean Architecture, Repository Pattern, and Client-Server (Nayan 2023; Fowler 2023; John Terra 2023).

`Clean architecture` gives us separation of concern and greater modularity. We achieve this by having our code split into 4 C# projects:

- `Shared`
- `Infrastructure`
- `Client`
- `Server`

`Shared` is for our data transfer objects (`DTO's`) and repository interfaces, so we will be able to mock our repositories without a dependency on them. `Infrastructure` is for the repositories, database context, and models for the database schema, where the repositories depend on `Shared`. `Client` contains the `Blazor` files responsible for the presentation layer and user interface. `Server` is the actual server that contains the API controllers, application configuration, and serves the frontend. The controllers depend on `Infrastructure` and `Shared`, as the controllers need the repositories and the DTO's. For the `Server` to serve the frontend it depends on the `Client`. Thus none of our projects are mutually dependent and `Shared` is the core, from where it aggravates outward to a layer containing `Infrastructure` and `Client`. Lastly a layer containing `Server` as the part that other systems interacts with.

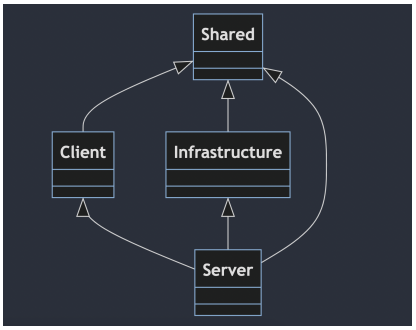

Figure 1: Project Architecture

We use a `Repository pattern` to add abstraction around our communication with the database. Even though we use EF Core which some would say is enough abstraction, then repositories allow us to make a switch over to use something like `Dapper` instead of `EF Core` (StackExchange 2023; Microsoft 2023c).

1.1.1 API Request Route

The route of data in our application starts when a user sends a request to the server. It will then pass through a few places before it returns. Below is a sequence diagram illustrating the journey of an arbitrary request along its designated 'route'.

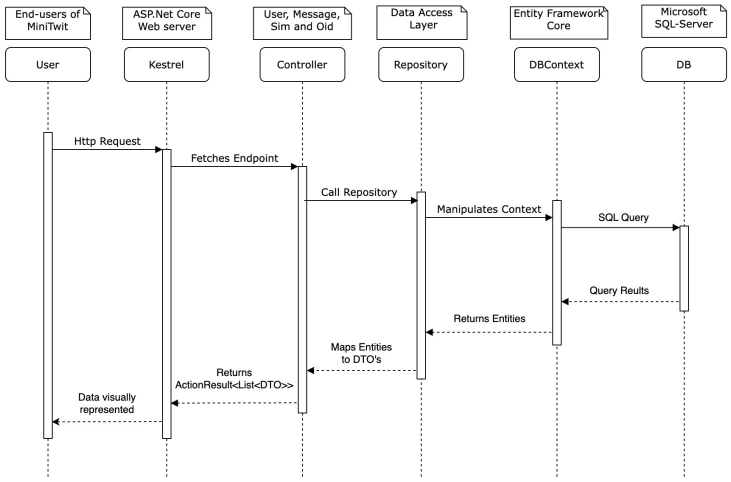

Figure 2: Data journey

After a user sends a request, the server (Kestrel) receives it and sends it to the controller method that matches the endpoint that was targeted. The controller then uses one of the repositories that asks EF Core to format a query and send it to the database. The database then returns the data in the opposite sequence, until it reaches the user again.

1.1.2 Database

For communication between server and database, we chose the popular open-source O/RM Entity Framework Core, also known as EF Core, that makes it possible to define the database schemas and maintain the database state. Additionally, it removes the type barriers between code and database. This makes it possible to query the database without writing SQL, which blocks SQL injection.

1.1.3 Dependencies

At a low abstraction level, our application has specific dependencies such as libraries and packages. These are imported and installed using dotnet as the NuGet package manager. A visual representation can be seen in figure 3.

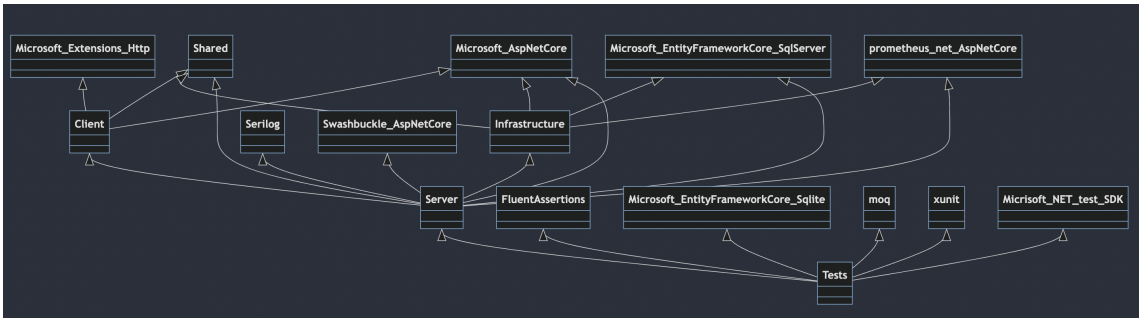

Figure 3: Package dependency graph for C# application

1.2 Tools & Expanded System Architecture

This section briefly describes the different tools used and their purpose. Most are expanded upon in the Process Perspective section. Figure 4 shows an overview of the infrastructure and dependencies of the system and its tools.

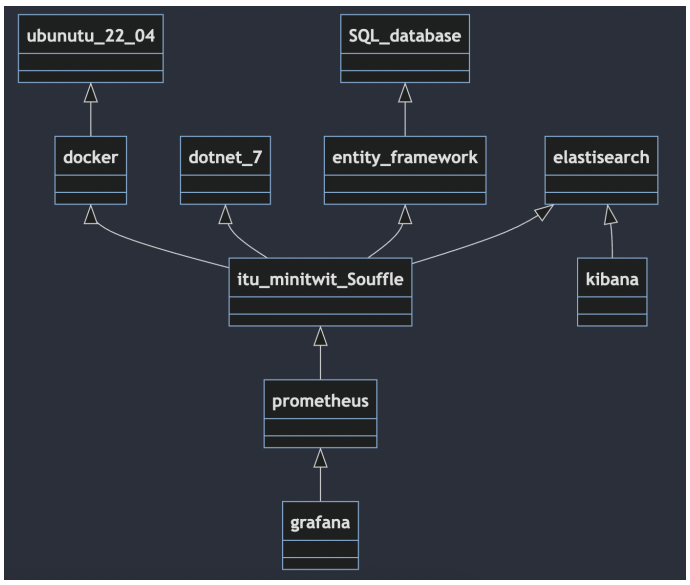

Figure 4: Infrastructure dependency graph

1.2.1 .NET 7

This is the platform our project is built on. We chose this runtime since it's the one we are most familiar with and it contains a lot of libraries that suit our needs.

1.2.2 Docker & Ubuntu 22.04

Docker is responsible for building, shipping, and running our application through containers (Docker 2023a). It makes it possible to have complete control of the environment in which the programs must run (we use Ubuntu 22.04). This ensures compatibility and portability; all it needs to run is the docker engine. Docker Swarm also enables scaling of the system within virtual machines (Docker 2023b).

1.2.3 Digital Ocean

As recommended, Digital Ocean is our chosen provider of cloud infrastructure. It's user-friendly and makes controlling our domain name simple by allowing it to be routed through Digital Oceans NS records (DigitalOcean, LLC. 2023). However, as providers go, it's expensive. We had limited free credits and were restricted in the number of virtual machines we could afford. Our system has two droplets; one for our server, and one for all our tools. We would have preferred to separate our tools on individual droplets, potentially managed as a swarm for easy balancing.

1.2.4 GitHub Actions

GitHub Actions allows us to integrate our workflows, and apply our tools and tests to our branches before merging them together, ensuring a certain level of quality control (GitHub 2023b).

1.2.5 Prometheus & Grafana

Prometheus is responsible for monitoring our system by periodically querying its specified metric endpoints for data (Prometheus 2023). This data is then accessed by Grafana, which is the platform responsible for visualizing it on its dashboards (Grafana Labs 2023). See 2.4.1 for more detailed information.

1.2.6 Superlinter & Snyk (Static Analysis Tools)

Lint Code Base (SuperLinter) is used to identify and reduce duplicated code segments and long methods since these smells may affect the maintainability and readability of the code (GitHub 2023f; Snyk 2023). It was chosen for its versatility, as it includes pre-configured linters for multiple languages and allows easy customization to align with our project's specific requirements.

Snyk is used to identify and address security vulnerabilities, by scanning open-source libraries and dependencies, which ensures robustness and security.

1.2.7 ElasticSearch & Kibana

ElasticSearch is a search and analytics engine that enables accurate and fast searching of data for retrieval and analysis, which makes it perfect for reading aggregated system logs (Elastic 2023a). Despite its scalability and distributability, we operate only a single instance, sacrificing redundancy. Kibana is a data visualization and exploration tool with dashboards to present data (Elastic 2023b).

Together these tools store, search, analyze, and visualize log data in our MiniTwit system. See 2.4.2 for more details.

1.3 Reflection of System State

DevOps is a crammed course. It has not been possible to implement everything. These are the tools the current system does not have, either because it was skipped, down-prioritized, or seemingly incompatible with our setup.

1.3.1 Terraform

This was never set up. We had a hard time understanding how the system should be set up and what parts of our infrastructure it should manage. Our rough understanding is that it could have made it easy to switch cloud providers and scale the number of machines the system should be distributed on (HashiCorp 2023).

1.3.2 NGINX in the Swarm

We have made a script that allows us to pass in a few addresses of VM's, and then have our Server deployed with an NGINX load balancer in front (Nginx 2023). We have had some issues with loading times when going through the load balancer to the servers.

1.3.3 Sonarqube & Code Climate

We never set up SonarQube and CodeClimate in our project due to time limitations (SonarSource 2023; Code Climate 2023). Theoretically, these tools would detect bugs, vulnerabilities, and code smells in order to improve code quality. Since we already implemented the Superliner and Snyk they were down-prioritized.

1.4 License

We chose the GNU General Public License v.3.0 (GPL-3.0) (*GNU General Public License v3.0* 2023). As it's more restrictive in future third-party usage than the MIT license, while remaining compatible with our tools, and promoting collaboration with the development community (*MIT License* 2023).

- Docker, Snyk, Grafana, and Prometheus are distributed under the Apache License 2.0, which is compatible with GPL-3.0.
- Kibana and ElasticSearch are licensed under the Elastic License, which has some limitations that we are in accordance with.
- .NET and Superliner are under the MIT License, which is compatible with GPL-3.0.
- NGINX is available under the NGINX Open Source License, which is compatible with GPL-3.0.
- Ubuntu is itself released under GPL.
- Identity Server has an open and free license while we don't sell our software or our services.
- Terraform is under the MPL 2.0 license, which is compatible as long as we do not distribute their source code.

2 Process' Perspective

2.1 Collaborative Development & Team Organization

At group formation, the team aligned expectations. Each member expected to spend 8-12 hours a week on the course, resulting in this schedule:

- Tuesday, 10-14: Project Work
- Tuesday, 14-16: Lecture
- Tuesday, 16-18: Exercises/Project Work
- Friday, 10-15: Project Work

All sessions were in person. Each member had the freedom to do additional work. Communication was through Discord (Discord, INC. 2023).

When working, we would at times do versions of pair programming. Teamwork was however difficult, since all concepts were new to everybody, and a lot of the understanding and progress came from trial and error. All sessions were an open forum where questions were welcome, and help was always possible to get. Sadly, we were not always good at remembering co-authorship on commits which might make the repository contributions look skewed.

We intended to utilize GitHub Issues to share the tasks between us, making sure not to be wasting time on duplicate work (GitHub 2023d).

2.2 CI/CD Chain

In this project, we have implemented a CI/CD chain by defining a handful of `.yml` files for GitHub Actions. A visual representation can be seen at figure 5:

- github-super-linter
- dotnet-Build-and-Test
- container-snyk
- continuous-deployment
- dotnet-format
- Scheduled-release
- latex-build

We also configured Dependabot, that scans for outdated dependencies and informs us of updates through pull requests (GitHub 2023a).

Dotnet-build-and-Test, container-snyk and dotnet-format run simultaneously every time a pull request is created. As part of our workflow, no pull requests were supposed to be merged into main without having a developer review it first. This was not entirely adhered to, however.

Every push to feature branches triggered the Super-linter. The first thing that happens when a pull request is created is that the application is built. We only build the application after pull requests, to avoid running unnecessary build cycles, reducing overhead. The dotnet-format workflow then operates on the build to guarantee a greater level of code consistency. Our build workflow then proceeds by checking out the branch, restoring the dependencies, building the application and executing the unit tests. This makes the test-stage dependant on the build-stage; if the build fails, the tests are never run. Depending on the test-suite, stability is ensured by making sure changes are tested, and old functionalities do not break (regression testing) (Hamilton 2023).

The Snyk workflow does security scans on the repository and notifies us of any vulnerabilities in open-source dependencies. Additionally, to mitigate the risk of inadvertently exposing secrets, we

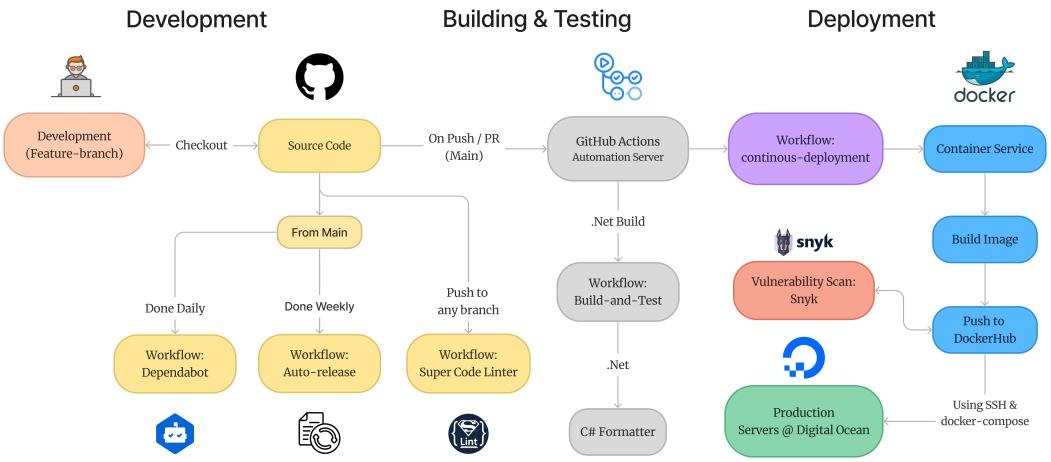

Figure 5: CI/CD tech-stack

used a Snyk feature on our code base to scan for sensitive data like connection strings, secret tokens, API keys, etc.

If either the building or testing fails, the pull request will be rejected.

The continous-deployment workflow has the purpose of automatically deploying our services. It builds the containers and connects to the cloud droplets and deploys.

Combined with the use of GitHub Secrets, these workflows provide a powerful solution to automated deployment (GitHub 2023e). It's a convenient way to manage our application's deployment process, while maintaining confidentiality and flexibility. This solution does however make scalability difficult because of hardcoded IP-addresses of our servers. This makes any changes tedious, as they would require manual changes to workflows and secrets.

The latex-build workflow only runs when files changed include those in the Report folder. It compiles the components of the report to a .pdf.

The auto-release workflow was implemented to further streamline the deployment process. It ensures that the latest changes are automatically released, scheduled for each Sunday at 20:00 CEST.

2.3 Repository Organization & Development Strategy

2.3.1 Repository Organization

We structured our code in a mono-repository for two main reasons:

1. It provides improved collaboration, as all developers are working on the same codebase.

2. It simplifies version control as we only have a single history for all code.

Another consideration was, that we didn't feel the need to divide it up, as the application is still relatively small. It would also add what we deemed to be unnecessary complexity by requiring different repositories to be cloud-hosted for cross-access since the necessary references in our C# code would no longer reside in the same .NET solution.

2.3.2 Development Strategy

We utilized feature branching which adds security gates from a require-to-pass test suite run in GitHub Actions and code reviews from other devs (GitHub 2023b). Main was intended to be secured by only allowing pushes through pull requests, but we allowed direct pushes as setting up secrets and editing workflows proved time-consuming otherwise.

Our tasks were organized in GitHub Issues. This made it possible for each member to always be able to see which tasks were up for grabs, and who were working on what. It also acted as a task backlog giving an overview of the work that was yet to be done.

As the project progressed, however, the tasks began including the whole group, and the Issues board became slightly irrelevant.

2.4 Monitoring & Logging

2.4.1 Monitoring

The monitoring of the system is powered by Prometheus which gathers metrics from endpoints created by the Prometheus.net NuGet package (Prometheus 2023). Metrics are then visualized by Grafana which queries Prometheus (Grafana Labs 2023). Though Grafana is available to anyone, you would need our login to access the metrics and dashboards. Our current metrics are:

- Http-requests (Duration)
- Total Users
- Total Messages
- Average request duration (Last 2 minutes)

Together, these metrics provide information about how much stress we can expect our system to experience, and what kind of request intensity we can expect. It also provides an overview of user growth, giving us a sense of the direction our service is headed.

Additionally, we are also making use of Digital Ocean to monitor our hardware such as the CPU and memory usage of our droplets.

2.4.2 Logging

Our projects logging is accomplished using the Serilog library, which is a popular logger for C# (Serilog Contributors 2023). We send the logs to ElasticSearch which stores and indexes them, allowing for fast and efficient lookups (Elastic 2023a). We use Kibana as a UI to view the log files (Elastic 2023b).

Serilog provides the ability to have multiple sinks, which are output destinations. We have a sink to ElasticSearch and the console of the container.

To tie Serilog and ElasticSearch together with Kibana, we made a docker-compose file responsible for setting up the connection and mounting the data from the container. Kibana then retrieves the logging data from ElasticSearch and displays it.

2.5 Security Assessment

Our security assessment consisted of a penetration test - Zed Attack Proxy, or ZAP, that showed some flaws in our program (OWASP 2023). Most seemed to be relatively simple to fix, and none seemed to be too severe in nature:

- No Anti-CSRF tokens were found in an HTML submission form
- Passive (90022 - Application Error Disclosure)
- Content Security Policy (CSP) Header Not Set
- Missing Anti-clickjacking Header
- X-Content-Type-Options Header Missing
- Hidden File Found
- Vulnerable JS Library

- XSLT Injection might be possible.
- Cloud Metadata Potentially Exposed

These items resulted in GitHub Issues; some of which were dealt with while others were down-prioritized.

2.6 Scaling & Load Balancing

Our applied strategy for scaling and load balancing involved utilizing Docker Swarm for fault tolerance and easy container management, together with Nginx working primarily as a load balancer and reverse proxy (Docker 2023b; Nginx 2023). Docker Swarm uses a cluster of nodes (workers and managers) and allows us to automatically adjust the number of running Docker containers based on the current workload. Another important feature of the swarm is that it automatically evenly distributes incoming requests, across the worker containers running the same service (internal load balancing with an ingress mesh).

The swarm has a maximum capacity limited to the combined resources (CPU, memory, disk space) of the server nodes within the swarm clusters. We used two servers for our application swarm and one server for the load-balancing swarm.

The first swarm consists of a server with manager nodes and a server with worker nodes running containers with our MiniTwit application. The second swarm includes a single manager node responsible for load balancing and reverse-proxying with Nginx. We designed it this way to separate concerns and allow the two swarms to scale independently. Combining Docker Swarm's automatic and dynamic scaling with Nginx's reverse-proxy capabilities resulted in an improvement in the system's availability and reliability.

By having Nginx handle and redirect requests to our service, we could eliminate the 'single point of failure', and decrease the impact a server crash has on our system. In case the primary replica node running our swarm was flooded or broke down, we could redirect the traffic from that server node to another upstream server, while docker swarms dynamical scaling would ensure that the system would be able to keep up with accelerating requests and users.

However, in our setup, there were a couple of things that could be improved. Initially, we only tested our system using a single instance of Nginx, which essentially only shifted the single point of failure from the application service to the load balancer in the Swarm. This decision involved weighing the trade-offs between reliability, complexity, and also the financial costs of creating more droplets. Also, we encountered some non-negligible internal network delays that affected the system's performance. When Nginx was responsible for the load balancing, requests sometimes took a long time to complete or re-direct. Due to the way Nginx was implemented, it didn't function as originally intended.

2.7 AI Assistance

As developers, we have embraced the tool that is AI. We have used ChatGPT as a sparring partner whenever we got a task we did not know how to solve, or when we got stuck (OpenAI 2023). It has mostly been good at providing ideas or starting points. But where it really shines is when troubleshooting or debugging. If nothing else, it has provided the services of a rubber duck (Wikipedia contributors 2023).

We have also used it in the formulation of documents like the SLA agreement, as it's mostly boilerplate text anyways. We provided it with some information about our system and what guarantees we could give, and it returned a rough draft we could finalize.

Finally, one team member has been using GitHub Copilot (GitHub 2023c) as an advanced intellisense tool, but not for making several lines of code. Meaning that it was used to finish individual lines of code, but not for making entire methods or blocks of code.

3 Lessons Learned Perspective:

3.1 Biggest Issues

The following section contains reflections on some of the biggest issues we encountered during this project.

3.1.1 Refactoring

It was recommended to refactor MiniTwit gradually, to avoid unnecessary work and mitigate some of the workload. We chose to ignore this recommendation and do a full refactor to C# from the beginning. This made for two workload-heavy weeks, but we succeeded which made the following weeks easier. This would most likely have been a mistake if the project was bigger than it was or if the codebase was one we had less experience with.

3.1.2 E2E Testing

During the initial phase of Evolution & Refactoring, our main issue was end-to-end (E2E) testing. Because we used WebAssembly, we had some issues rendering the pages correctly which made asserting anything from the tests difficult. Implementing E2E testing is a very important part of developing software. The lesson learned in this project is how valuable E2E testing can be in order to detect bugs and prevent technical debt; and not doing this hurt us later in the project. For example, we had problems with our `follows` endpoint - an issue we didn't realize we had until late in the project.

3.1.3 Database Attack

On the first days of operations, our database experienced an unsuccessful bruteforce login attack, that shut down the database. Identifying the problem took time due to our limited experience in accessing container logs. However, a team member who had taken the Security course was able to promptly identify the issue by examining these logs. After it happened the second time, we made the decision to add a firewall on Digital Ocean's network. This experience taught us a valuable lesson about the presence of malicious agents online, and the importance of considering their actions when maintaining a system.

3.1.4 Implementing Scalability

After several attempts at using Terraform, an infrastructure-as-code (IaC) tool, we discovered the challenges associated with deployment with a new infrastructure environment. It was something we wanted to implement so we could create, modify and destroy infrastructure in an automated way. After some reflection on our current capacity for scalability, we've come to the conclusion that we could benefit greatly from something like Terraform. Our current set-up relies on manual entries of Github-secrets, and hard-defined docker-compose files for deploying to our architecture. This is an area in which we could greatly improve the scalability of our system.

3.2 Our DevOps Style of Working

The purpose of this course and this project has been to explore the DevOps way of working. MiniTwit is the core this exploration has been built around. As a software project, it resembles those we as developer-students have encountered previously, where the focus has been on the gradual implementation of features. But in this course, the focus has instead been on the utilization of tools and practices that simplify monitoring and maintenance, and automate releases and deployment. These tools and this way of working have provided us with a new perspective on how a project can be structured and managed. It has taught us how certain practices and workflows can act as a catalyst for productivity and quality assurance. With DevOps in the toolbox, there are new considerations to make at the start of every new project.

4 Appendix

4.1 Diagrams

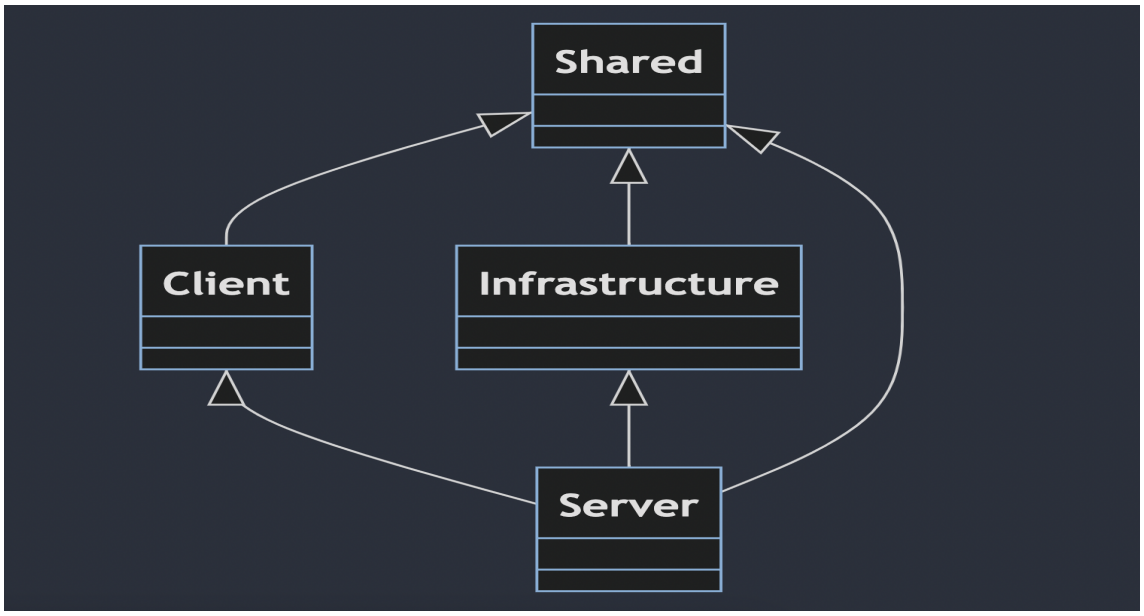

Figure 6: Larger version of the project dependency graph

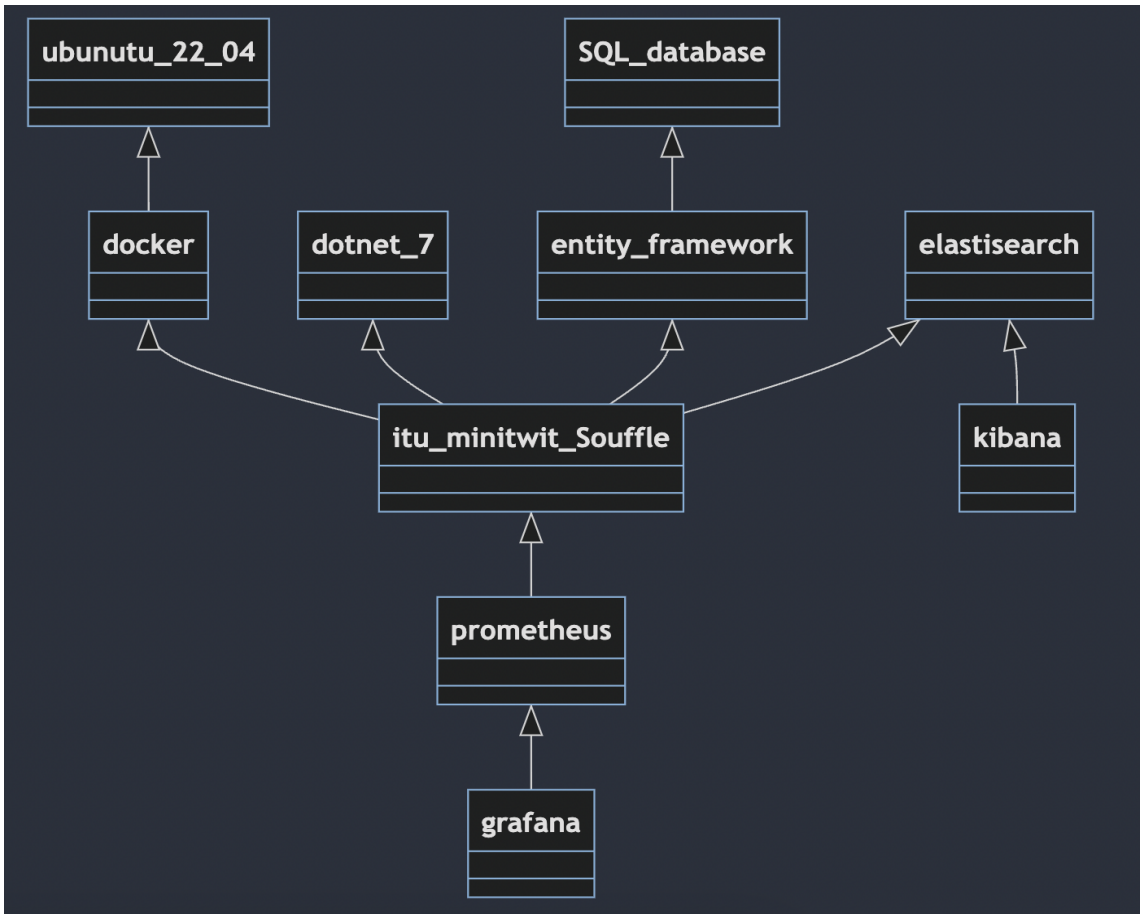

Figure 7: Larger version of the application dependency graph

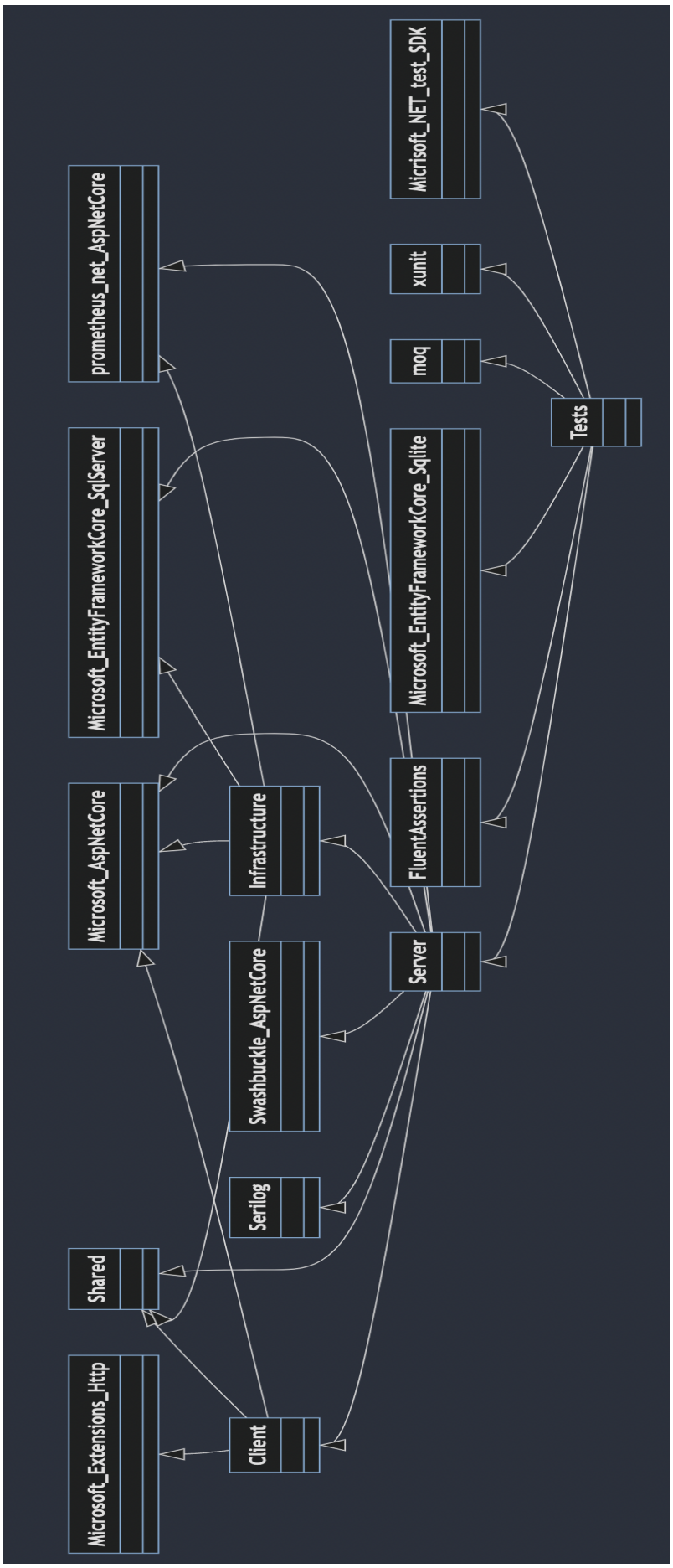

Figure 8: Larger version of Package dependency graph for C# application

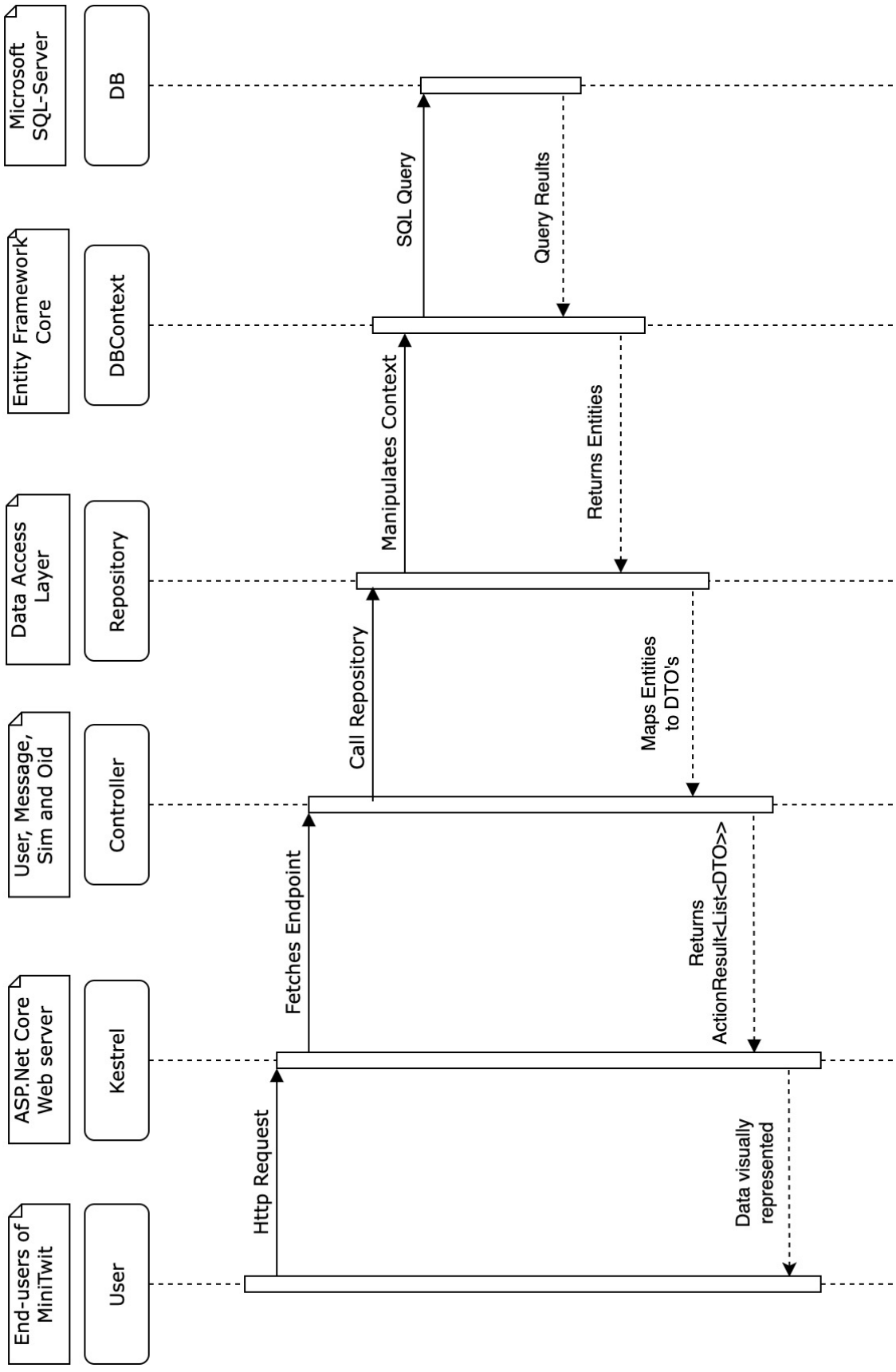

Figure 9: Larger version of API sequence diagram

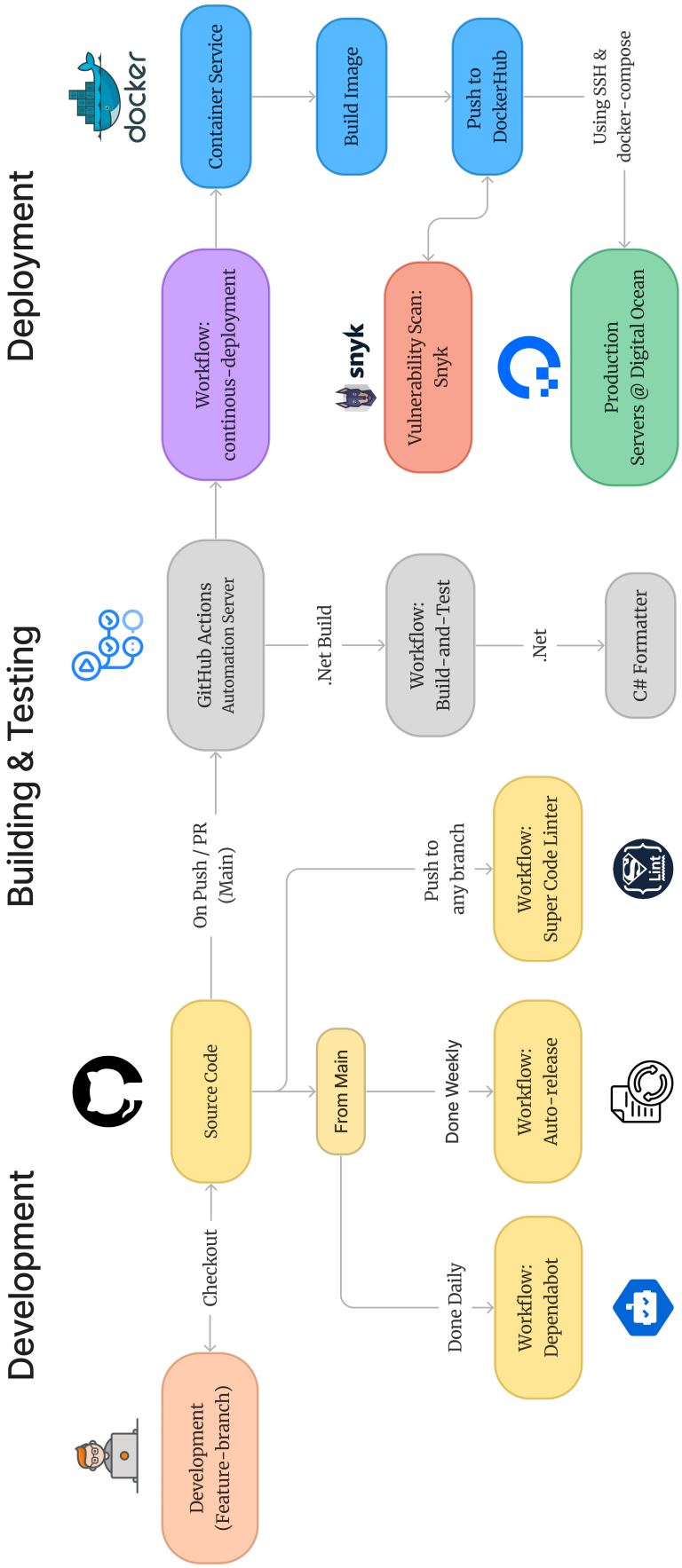

Figure 10: Larger version of CI/CD tech-stack

References

Code Climate (2023). *CodeClimate*. Code Climate, Inc. URL: https://codeclimate.com/ (visited on 05/23/2023).

DigitalOcean, LLC. (2023). *Digital Ocean*. DigitalOcean, LLC. URL: https://www.digitalocean.com/ (visited on 05/23/2023).

Discord, INC. (2023). *Discord*. Discord, INC. URL: https://discord.com/ (visited on 05/23/2023).

Docker (2023a). *Docker*. Docker. URL: https://www.docker.com/ (visited on 05/23/2023).

– (2023b). *Docker Swarm*. Docker. URL: https://docs.docker.com/engine/swarm/ (visited on 05/23/2023).

Elastic (2023a). *Elasticsearch*. Elastic. URL: https://www.elastic.co/elasticsearch (visited on 05/23/2023).

– (2023b). *Kibana*. Elastic. URL: https://www.elastic.co/kibana (visited on 05/23/2023).

Fowler, Martin (2023). *Repository*. Martin Fowler. URL: https://martinfowler.com/eaaCatalog/repository.html (visited on 05/23/2023).

GitHub (2023a). *Dependabot*. GitHub, Inc. URL: https://github.com/dependabot (visited on 05/23/2023).

– (2023b). *GitHub Actions*. GitHub. URL: https://docs.github.com/en/actions (visited on 05/23/2023).

– (2023c). *GitHub Copilot*. GitHub. URL: https://copilot.github.com/ (visited on 05/23/2023).

– (2023d). *GitHub Issues*. GitHub. URL: https://docs.github.com/en/issues (visited on 05/23/2023).

– (2023e). *GitHub Secrets*. GitHub. URL: https://docs.github.com/en/actions/reference/encrypted-secrets (visited on 05/23/2023).

– (2023f). *SuperLinter*. GitHub. URL: https://github.com/github/super-linter (visited on 05/23/2023).

GNU General Public License v3.0 (2023). [Online]. Version 3.0. Free Software Foundation. URL: https://www.gnu.org/licenses/gpl-3.0.en.html (visited on 05/23/2023).

Grafana Labs (2023). *Grafana*. Grafana Labs. URL: https://grafana.com/ (visited on 05/23/2023).

Hamilton, Thomas (2023). *Regression Testing*. Guru99. URL: https://www.guru99.com/regression-testing.html (visited on 05/23/2023).

HashiCorp (2023). *Terraform*. HashiCorp. URL: https://www.terraform.io/ (visited on 05/23/2023).

John Terra (2023). *What is Client-Server Architecture?* Simplilearn. URL: https://www.simplilearn.com/what-is-client-server-architecture-article (visited on 05/23/2023).

Microsoft (2023a). *.NET*. Microsoft. URL: https://dotnet.microsoft.com/ (visited on 05/23/2023).

– (2023b). *ASP.NET Blazor*. Microsoft Corporation. URL: https://dotnet.microsoft.com/en-us/apps/aspnet/web-apps/blazor (visited on 05/23/2023).

– (2023c). *Entity Framework Core*. Microsoft Corporation. URL: https://docs.microsoft.com/en-us/ef/core/ (visited on 05/23/2023).

MIT License (2023). [Online]. Massachusetts Institute of Technology. URL: https://opensource.org/licenses/MIT (visited on 05/23/2023).

Nayan, Bishwanath Dey (2023). *Introduction to Clean Architecture and Implementation with ASP.NET Core*. C# Corner. URL: https://www.c-sharpcorner.com/article/introduction-to-clean-architecture-and-implementation-with-asp-net-core/ (visited on 05/23/2023).

Nginx (2023). *Nginx*. Nginx. URL: https://nginx.org/ (visited on 05/23/2023).

NuGet (2023). *NuGet Gallery*. Microsoft Corporation. URL: https://www.nuget.org/ (visited on 05/23/2023).

OpenAI (2023). *ChatGPT*. OpenAI. URL: https://openai.com/product/chatgpt (visited on 05/23/2023).

OWASP (2023). *Zed Attack Proxy (ZAP)*. OWASP. URL: https://owasp.org/www-project-zap/ (visited on 05/23/2023).

Prometheus (2023). *Prometheus*. Prometheus. URL: https://prometheus.io/ (visited on 05/23/2023).

Serilog Contributors (2023). *Serilog*. Serilog. URL: https://serilog.net/ (visited on 05/23/2023).

Snyk (2023). *Snyk*. Snyk Ltd. URL: https://snyk.io/ (visited on 05/23/2023).

Software, Duende (2023). *ASP.NET IdentityServer*. Duende Software LLC. URL: https://docs.duendesoftware.com/identityserver/v6/quickstarts/5_aspnetid/ (visited on 05/23/2023).

SonarSource (2023). *SonarQube*. SonarSource SA. URL: https://www.sonarqube.org/ (visited on 05/23/2023).

StackExchange (2023). *Dapper*. Stack Exchange Inc. URL: https://dapperlib.github.io/Dapper/ (visited on 05/23/2023).

Wikipedia contributors (2023). *Rubber Duck Debugging*. Wikipedia. URL: https://en.wikipedia.org/wiki/Rubber_duck_debugging (visited on 05/23/2023).