# BAB II
# TINJAUAN PUSTAKA

## 2.1. Tinjauan Pustaka

### 2.1.1. Implementasi Arsitektur Microservices pada Online Judge

Penerapan arsitektur microservices pada sistem Online Judge telah banyak diteliti sebagai solusi untuk mengatasi keterbatasan skalabilitas dan pemeliharaan pada arsitektur monolitik konvensional. Pan dkk. mengusulkan sistem cloud-native yang menerapkan Domain-Driven Design (DDD) dan Event-Driven Architecture menggunakan RabbitMQ serta MongoDB untuk menangani penjadwalan sumber daya CPU dan GPU secara dinamis [6]. Pendekatan serupa dalam pemisahan layanan dilakukan oleh Liang dan Tu, yang membangun platform untuk kebutuhan pengajaran pemrograman menggunakan ekosistem Spring Cloud (Nacos, Feign, Gateway) untuk memisahkan modul pengguna, soal, dan kontes guna meningkatkan efisiensi koreksi tugas [5]. Sementara itu, Yu dkk. berfokus pada aspek keamanan dan penanganan konkurensi tinggi dengan memisahkan judge machine sebagai layanan mikro mandiri berbasis Docker yang dikelola melalui mekanisme load balancing menggunakan Haproxy dan pemantauan sumber daya dengan Telegraf dan Consul [11].

Meskipun ketiga penelitian tersebut telah berhasil mendekomposisi sistem monolitik, penelitian tugas akhir ini memiliki relevansi dan perbedaan spesifik dalam pendekatan teknisnya. Penelitian ini berbeda dengan [5] yang lebih menekankan pada fitur manajemen edukasi, karena lebih berfokus pada ketahanan sistem terhadap lonjakan trafik dengan mengadopsi mekanisme antrean pesan. Jika dibandingkan dengan [11] yang masih mengandalkan Haproxy untuk load balancing berbasis HTTP, penelitian ini menawarkan kebaruan melalui implementasi distributed online judge di mana distribusi beban kerja diatur sepenuhnya melalui antrean pesan untuk mencegah blocking process. Dengan demikian, penelitian ini mensintesis mekanisme isolasi sandbox berbasis kontainer dari [11] serta mengadaptasi arsitektur event-driven menggunakan RabbitMQ dari

[6] untuk memfasilitasi komunikasi asinkron antara layanan submisi dan penilaian, ke dalam ekosistem Spring Boot guna menciptakan platform coding contest yang skalabel dan adil.

### 2.1.2. Teknologi Kontainerisasi dan Orkestrasi

Penerapan teknologi virtualisasi ringan atau kontainerisasi telah menjadi standar baru dalam pengembangan sistem Online Judge untuk mengatasi inefisiensi arsitektur berbasis mesin virtual. Han dan Zhang [12] mengembangkan sistem eksperimental berbasis Docker yang menawarkan lingkungan isolasi efisien untuk pengajaran pemrograman, menggantikan penggunaan mesin virtual yang memakan banyak sumber daya. Pendekatan serupa diterapkan oleh Ariff [13] yang secara spesifik memanfaatkan Docker sebagai mekanisme sandboxing untuk mengeksekusi kode aplikasi berbasis jaringan yang tidak tepercaya, memastikan bahwa eksekusi kode berbahaya tidak dapat merusak sistem operasi host. Lebih lanjut, Kamod et al. [7] mengintegrasikan teknologi kontainerisasi ini dengan orkestrator Kubernetes, yang terbukti mampu menangani volume submisi tinggi secara aman dan skalabel melalui manajemen sumber daya otomatis.

Analisis literatur menegaskan bahwa kontainerisasi merupakan solusi efektif untuk menjamin keamanan dan isolasi eksekusi kode. Kendati demikian, implementasi pada penelitian terdahulu [13] [12] cenderung masih bersifat eksperimental dan rentan mengalami penurunan performa pada beban tinggi. Meskipun Kamod et al. [7] telah menawarkan solusi skalabilitas melalui Kubernetes, penelitian dalam tugas akhir ini melengkapi pendekatan tersebut dengan mengintegrasikan Message Broker dalam arsitektur microservices terdistribusi. Sistem yang diusulkan tidak hanya mengandalkan isolasi kontainer untuk aspek keamanan, tetapi juga menerapkan komunikasi asinkron untuk memisahkan antrean submisi dan proses penilaian (judging). Pemisahan ini bertujuan untuk mencegah terjadinya blocking process, sehingga responsivitas sistem tetap terjaga dalam skenario kompetisi.

### 2.1.3. Skalabilitas dan Arsitektur Berbasis Event-Driven

Berbagai penelitian terdahulu menyoroti pentingnya arsitektur sistem yang mampu tetap stabil saat menangani beban kerja tinggi. Watanobe et al. [14] memaparkan arsitektur Aizu Online Judge yang memanfaatkan Load Balancer untuk mendistribusikan tugas ke banyak server penilaian, sebuah pendekatan yang terbukti menjaga ketersediaan sistem hingga 99% selama satu dekade operasi. Sejalan dengan itu, Došilović dan Mekterović [1] mengusulkan agar sistem eksekusi kode (OCES) dipisahkan menjadi layanan mandiri yang dapat diperbanyak jumlahnya (horizontal scaling) untuk mempercepat pemrosesan antrean. Secara lebih spesifik, Arvon et al. [15] menerapkan Event-Driven Architecture (EDA) pada sistem grader berbasis microservice. Dalam implementasi ini, message broker berfungsi menjembatani layanan backend dan grader secara asinkron, sehingga proses penerimaan kode dari pengguna tidak akan macet (blocking) meskipun proses penilaian di sisi lain sedang memakan banyak sumber daya komputasi.

Berdasarkan tinjauan terhadap penelitian-penelitian di atas, terlihat jelas bahwa penggunaan komunikasi asinkron sangat krusial untuk menjaga responsivitas aplikasi online judge. Penelitian tugas akhir ini mengadopsi prinsip memperbanyak unit pekerja (horizontal scaling) seperti saran Došilović et al. [1] serta manajemen distribusi beban antrean yang dibahas oleh Watanobe et al. [14]. Namun, penelitian ini membedakan diri dengan berfokus pada ekosistem kompetisi pemrograman yang menuntut penanganan konkurensi tinggi menggunakan RabbitMQ. Jika Arvon et al. [15] lebih menitikberatkan pada validasi fungsi penilaian untuk kebutuhan akademis, penelitian ini memperluas konsep tersebut dengan mengintegrasikan mekanisme isolasi berbasis kontainer yang terdistribusi. Hal ini bertujuan untuk menjamin stabilitas latensi sistem dan keamanan eksekusi kode saat menghadapi lonjakan trafik yang ekstrem dalam situasi kontes nyata..

### 2.1.4. Sistem Manajemen Kontes dan Implementasi Skala Nasional

Implementasi sistem manajemen kontes skala nasional telah didokumentasikan di berbagai negara untuk mendukung kompetisi pemrograman yang masif. Laaksonen dan Talvitie mengembangkan Code Submission Evaluation

System (CSES) untuk Olimpiade Informatika Finlandia, yang memisahkan antarmuka pengguna berbasis PHP dari klaster judge servers terdistribusi yang memanfaatkan utilitas Isolate untuk eksekusi kode aman [16]. Di sisi lain, Hernández González dkk. mengadopsi Don Mills Online Judge (DMOJ) untuk Olimpiade Informatika Kuba, menggunakan arsitektur yang menghubungkan aplikasi web Django dengan judge workers melalui komponen bridge untuk menangani ratusan pengguna secara simultan [17]. Tsedevsuren dkk. juga melaporkan penggunaan Contest Management System (CMS) pada Olimpiade Nasional Mongolia, di mana sistem tersebut diintegrasikan dengan portal registrasi eksternal dan menggunakan load balancer untuk mendistribusikan beban kerja layanan web saat kontes berlangsung [18].

Kajian terhadap platform-platform tersebut memperlihatkan bahwa pemisahan beban kerja antara antarmuka pengguna dan eksekusi penilaian adalah standar krusial, sebagaimana arsitektur terdistribusi yang diterapkan pada CSES [16] dan penggunaan bridge penghubung pada DMOJ [17]. Namun, penelitian ini mengajukan pengembangan lebih lanjut dari konsep distribusi tugas yang digunakan pada DMOJ dan CMS, yang terkadang masih bergantung pada konfigurasi server spesifik atau mekanisme komunikasi sinkron yang berpotensi blocking [17] [18]. Berbeda dengan pendekatan tradisional tersebut, tugas akhir ini menerapkan arsitektur Microservices yang sepenuhnya decoupled menggunakan Message Broker (RabbitMQ) untuk menjamin komunikasi asinkron yang lebih andal dibandingkan mekanisme antrean pada sistem terdahulu. Selain itu, jika CSES menekankan pentingnya isolasi keamanan menggunakan Isolate pada server fisik [16], penelitian ini mengadopsi konsep keamanan tersebut tetapi membungkusnya dalam teknologi kontainerisasi (Docker) untuk meningkatkan portabilitas dan efisiensi skalabilitas sistem.

## 2.2. Celah Penelitian

Berdasarkan tinjauan pustaka yang telah dilakukan, sejumlah penelitian terdahulu telah berupaya mengatasi tantangan skalabilitas pada sistem Online Judge dengan berbagai pendekatan. Liang dan Tu [5] serta Pan et al. [6] telah menerapkan

arsitektur microservices untuk memecah modul sistem guna memudahkan pemeliharaan. Akan tetapi, penelitian Liang dan Tu [5] lebih menitikberatkan pada manajemen kelas untuk kebutuhan edukasi dengan karakteristik beban trafik yang cenderung statis dan terprediksi. Konteks ini dinilai kurang relevan untuk menangani lonjakan trafik ekstrem yang terjadi secara tiba-tiba dalam sebuah kompetisi pemrograman. Di sisi lain, Došilović dan Mekterović [1] serta Kamod et al. [7] telah menawarkan solusi skalabilitas melalui penskalaan horizontal dan orkestrasi Kubernetes. Walaupun demikian, pendekatan tersebut sering kali masih menghadapi risiko proses yang terhambat atau blocking process pada level aplikasi karena mekanisme antrean yang belum dikelola secara asinkron sepenuhnya. Selain itu, implementasi keamanan berbasis kontainer pada penelitian Ariff [13] masih bersifat eksperimental dan belum terintegrasi secara utuh dengan sistem manajemen kontes yang menuntut pembaruan status penilaian secara langsung atau real-time kepada peserta.

Oleh karena itu, celah penelitian yang dieksplorasi secara spesifik dalam tugas akhir ini adalah pengembangan arsitektur backend untuk platform coding contest yang tidak hanya terdistribusi tetapi juga sepenuhnya mengadopsi komunikasi asinkron menggunakan Message Broker seperti RabbitMQ yang dipadukan dengan eksekusi terisolasi berbasis kontainer. Penelitian ini menjadi sangat penting karena arsitektur microservices standar tanpa mekanisme event-driven yang kuat sering kali gagal mempertahankan latensi rendah saat menghadapi ribuan submisi konkuren dalam detik yang sama yang merupakan karakteristik utama dari sebuah kontes pemrograman. Melalui integrasi RabbitMQ yang berfungsi sebagai penyangga beban antara layanan submisi dan layanan penilaian, penelitian ini bertujuan untuk menjamin bahwa antarmuka pengguna tetap responsif dan tidak mengalami timeout meskipun pekerja penilaian atau judge workers sedang bekerja pada kapasitas penuh. Penelitian ini juga memperkenalkan mekanisme *warm worker pool* yang memastikan judge workers selalu siaga guna mengatasi cold start problem, sehingga sistem tidak perlu membuat kontainer baru untuk setiap eksekusi dan waktu respons menjadi lebih cepat.

| Penulis | Message Broker | Framework | Arsitektur Sistem | Mekanisme Komunikasi | Lingkungan Isolasi | Fokus Penggunaan |
|---|---|---|---|---|---|---|
| Liang dkk. [5] | - | Spring | Microservices | RESTful API | Docker Container | Pengajaran Pemrograman dan Manajemen Kelas |
| Kamod dkk. [7] | - | Gin-Gonic | Microservices | RESTful API dan gRPC | Docker Container dan Kubernetes | Edukasi |
| Ariff [13] | - | MEAN Stack (MySQL, ExpressJS, AngularJS, NodeJS) | Modular Monolith | RESTful API | Docker Container | Edukasi |
| Arvon dkk. [15] | Ada | - | Microservices | Asynchronous Messaging | - | Penilaian di Lingkungan Akademik |
| Watanobe dkk.[14] | - | Spring dan Scala | Microservices | RESTful API | OS Script Sandboxing | Kompetisi Pemrograman |
| Došilović dkk. [1] | - | - | Microservices | Asynchronous dan RESTful API | Isolate | Edukasi, Rekrutmen, dan Kompetisi |
| Liang dkk.[19] | - | Go | Service Based Architecture | RESTful API | Docker Container dan Linux seccomp | Edukasi |
| Han dkk.[12] | - | Spring | Monolith | RESTful API | Docker Container | Edukasi |

| Penulis | Message Broker | Framework | Arsitektur Sistem | Mekanisme Komunikasi | Lingkungan Isolasi | Fokus Penggunaan |
|---|---|---|---|---|---|---|
| Yu dkk. [11] | - | Spring | Microservices | RESTful API | Docker Container | Edukasi |
| Pan dkk. [6] | Ada | - | Microservices | Asynchronous Messaging | Docker Container dan Kubernetes | Edukasi |
| Hernández González dkk.[17] | Ada | Django | Microservices | Asynchronous Messaging | Virtual Private Server (VPS) | Olimpiade Nasional Kompetisi Pemrograman |
| Audrito dkk. [4] | - | Rust, Angular, dan Typescript | Client Server Architecture | Websocket | - | Edukasi |
| Laaksonen dkk. [16] | - | PHP dan Rust | Microservices | RESTful API | Isolate | Kompetisi Pemrograman |
| Rahmatulloh dkk. [20] | - | VueJS dan AWS Lambda | Serverless Microservices | RESTful API | Judge0 API | Evaluasi kode dan deteksi plagiarisme |
| Tsedevsuren dkk. [18] | - | - | Microservices | - | Linux namespaces | Kompetisi Pemrograman |
| Penulis (2025) | Ada | React dan Spring Boot | Microservices | Asynchronous | Docker Container dan Isolate | Kompetisi Pemrograman |

**Tabel 2. 1. Tabel Perbandingan Penelitian**

## 2.3. Landasan Teori

### 2.3.1. Online Judge

Online Judge didefinisikan sebagai sistem perangkat lunak yang dirancang untuk menerima, mengompilasi, dan mengeksekusi kode sumber dari pengguna secara otomatis untuk menguji kebenarannya terhadap serangkaian *test cases* yang telah ditentukan sebelumnya. Proses evaluasi dalam sebuah Online Judge berjalan di bawah batasan-batasan yang ketat untuk memastikan keamanan. Sistem ini telah menjadi komponen infrastruktur vital dalam pendidikan ilmu komputer modern serta penyelenggaraan kompetisi pemrograman global karena kemampuannya dalam memberikan umpan balik instan dan objektif kepada pengguna tanpa intervensi manusia [14].

Secara fundamental, mekanisme penilaian pada Online Judge bekerja dengan cara membandingkan keluaran yang dihasilkan oleh kode pengguna dengan keluaran kunci jawaban yang diharapkan untuk setiap masukan yang diberikan. Laaksonen dan Talvitie menjelaskan bahwa proses ini harus dilakukan dalam lingkungan yang terkontrol di mana setiap eksekusi dibatasi oleh parameter waktu dan memori untuk menentukan efisiensi algoritma [16]. Hasil dari proses evaluasi ini kemudian dikembalikan kepada pengguna dalam bentuk status penilaian atau *verdict*, seperti *Accepted* (AC) jika jawaban benar, *Wrong Answer* (WA) jika keluaran salah, atau *Time Limit Exceeded* (TLE) jika eksekusi melebihi batas waktu yang diizinkan.

Dalam konteks kompetisi pemrograman berskala besar, Online Judge tidak hanya berfungsi sebagai penilai kode, tetapi juga harus memenuhi standar keandalan dan skalabilitas yang tinggi. Došilović dan Mekterović menekankan bahwa sistem harus mampu menangani lonjakan konkurensi submisi tanpa mengalami kegagalan layanan atau *downtime*, yang sering terjadi pada awal atau akhir waktu kontes [1]. Aspek keamanan menjadi prioritas utama karena sistem harus mengeksekusi *arbitrary code* dari pihak eksternal mengakibatkan diperlukannya mekanisme isolasi yang kuat untuk mencegah kode berbahaya merusak sistem *host* atau memanipulasi hasil penilaian. Kebutuhan akan

skalabilitas dan isolasi inilah yang mendorong evolusi arsitektur Online Judge dari model monolitik sederhana menuju arsitektur terdistribusi yang lebih kompleks.

### 2.3.2. *Message Broker*

*Message broker* merupakan perangkat lunak perantara yang memfasilitasi pertukaran data antar aplikasi, sistem, atau layanan yang berbeda secara efisien. Perangkat lunak ini bekerja dengan cara menerjemahkan pesan dari protokol pengirim ke protokol penerima, sehingga memungkinkan layanan-layanan yang tidak saling terhubung secara langsung untuk tetap berkomunikasi [21]. Dalam arsitektur *microservices*, keberadaan *message broker* sangat krusial untuk menciptakan sistem yang bersifat *loosely coupled*. Hal tersebut menandakan bahwa pengirim pesan tidak perlu mengetahui lokasi atau status aktif dari penerima pesan saat mengirimkan data, yang pada akhirnya meningkatkan fleksibilitas pengembangan sistem [21].

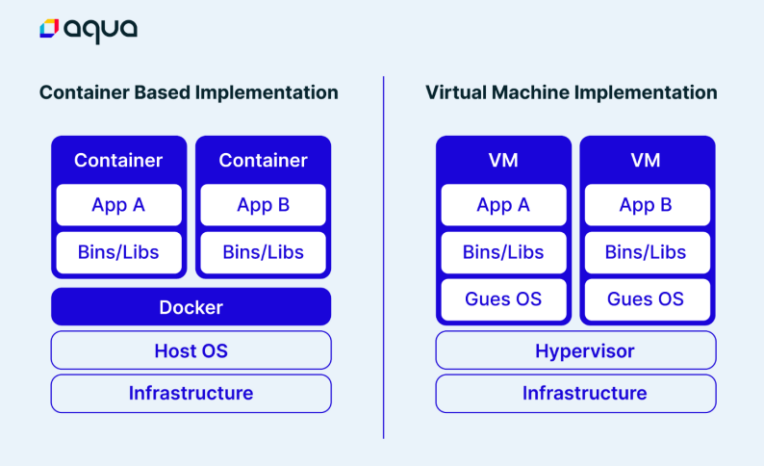
Penerapan *message broker* memungkinkan terjadinya pola komunikasi asinkron dalam sistem terdistribusi. Berbeda dengan komunikasi sinkron berbasis HTTP yang mengharuskan pengirim menunggu respons balik dari penerima sebelum melanjutkan proses, komunikasi asinkron memungkinkan proses pengiriman berjalan tanpa hambatan waktu tunggu. Arvon dkk. menjelaskan bahwa dalam konteks sistem penilaian otomatis, mekanisme ini mencegah terjadinya *blocking process* pada antarmuka pengguna ketika sistem sedang menangani beban komputasi yang berat [15]. Pesan atau tugas yang masuk akan ditampung sementara dalam antrean yang dikelola oleh broker hingga layanan pekerja siap untuk memprosesnya.

Salah satu implementasi populer dari teknologi ini adalah RabbitMQ yang berfungsi sebagai *buffer* untuk mendistribusikan beban kerja secara merata ke berbagai layanan konsumen. Penggunaan RabbitMQ menjamin reliabilitas sistem karena pesan yang belum terproses akan tetap tersimpan aman dalam antrean meskipun terjadi lonjakan lalu lintas data yang tiba-tiba. Karakteristik ini sangat relevan untuk platform kompetisi pemrograman yang sering mengalami lonjakan submisi dalam waktu singkat. Dengan memisahkan proses penerimaan submisi dan

proses penilaian, sistem dapat menjaga ketersediaan layanan tetap tinggi dan memberikan respons yang cepat kepada pengguna tanpa terbebani oleh durasi eksekusi kode di sisi backend.

### 2.3.3. Virtualisasi dan Kontainerisasi

Virtualisasi adalah teknologi yang memungkinkan pembuatan versi maya dari sumber daya komputer, seperti sistem operasi, perangkat penyimpanan, atau jaringan. Dalam model virtualisasi tradisional berbasis Virtual Machine (VM), sebuah hypervisor digunakan untuk meniru perangkat keras fisik sehingga beberapa sistem operasi tamu dapat berjalan di atas satu mesin fisik yang sama [22]. Meskipun metode ini memberikan isolasi yang kuat, setiap VM harus memuat sistem operasi penuh beserta kernelnya sendiri. Hal ini mengakibatkan penggunaan sumber daya yang boros dan waktu booting yang relatif lambat, sehingga menjadi kendala utama saat sistem perlu melakukan penskalaan cepat [13].




**Gambar 2. 1 Perbandingan Arsitektur Kontainer dan Mesin Virtual**

Sebagai solusi atas inefisiensi tersebut, teknologi kontainerisasi hadir dengan pendekatan yang lebih ringan. Berbeda dengan VM yang memvirtualisasi perangkat keras, kontainerisasi melakukan virtualisasi pada level sistem operasi. Teknologi ini memungkinkan beberapa aplikasi untuk berjalan di atas kernel host yang sama namun tetap terisolasi satu sama lain dalam ruang pengguna atau *user*

*space* yang terpisah. Karena tidak perlu memuat sistem operasi secara penuh, kontainer memiliki ukuran yang jauh lebih kecil dibandingkan VM dan dapat dijalankan dalam hitungan detik. Efisiensi ini menjadikan kontainerisasi pilihan ideal untuk arsitektur microservices yang menuntut fleksibilitas tinggi.

Docker merupakan platform perangkat lunak yang menjadi standar industri saat ini untuk mengimplementasikan teknologi kontainer. Docker menyediakan mekanisme untuk mengemas aplikasi beserta seluruh dependensinya seperti pustaka, *runtime*, dan konfigurasi sistem ke dalam sebuah unit standar yang disebut *image*. Pengemasan ini menjamin konsistensi lingkungan eksekusi, di mana aplikasi akan berjalan dengan perilaku yang sama persis baik di komputer pengembang maupun di server produksi. Dalam konteks *Online Judge*, Docker dimanfaatkan untuk menciptakan lingkungan sandbox yang aman dan terisolasi bagi setiap submisi kode peserta. Pendekatan ini memastikan bahwa eksekusi kode yang berpotensi berbahaya tidak dapat memengaruhi sistem utama, sekaligus menjamin keadilan penilaian karena setiap kode dijalankan dalam kondisi sumber daya yang identik.

### 2.3.4. *Sandboxing*

*Sandboxing* merupakan mekanisme keamanan perangkat lunak yang digunakan untuk menjalankan program atau kode dalam lingkungan terisolasi guna membatasi aksesnya terhadap sumber daya *host*. Dalam ekosistem *Online Judge*, mekanisme ini menjadi persyaratan mutlak karena sistem diwajibkan untuk mengeksekusi *arbitrary code* yang dikirimkan oleh pengguna dari sumber yang tidak tepercaya. Došilović dan Mekterović menegaskan bahwa tanpa penerapan isolasi yang ketat, kode tersebut berpotensi mengandung instruksi berbahaya yang dapat merusak sistem operasi, mengakses data sensitif, atau mengganggu integritas *software* [1]. Oleh karena itu, *sandbox* bertindak sebagai lapisan pertahanan yang membatasi ruang gerak kode peserta hanya pada area yang telah ditentukan.

Selain fungsi keamanan, penerapan sandboxing berperan vital dalam menjamin keadilan kompetisi melalui pengendalian sumber daya yang ketat. Sistem harus memastikan bahwa setiap submisi mendapatkan alokasi waktu pemrosesan

CPU dan memori yang konsisten, terlepas dari fluktuasi beban kerja yang sedang terjadi pada perangkat. Tanpa isolasi sumber daya yang baik, pengukuran waktu eksekusi menjadi tidak akurat dan rentan menyebabkan kesalahan penilaian seperti *Time Limit Exceeded* yang tidak valid bagi peserta [8]. Laaksonen dan Talvitie menambahkan bahwa pembatasan ini juga mencakup pelarangan akses terhadap *system calls* tertentu, seperti upaya membuka koneksi jaringan atau memanipulasi berkas sistem, untuk menjaga keamanan lingkungan penilaian [16].

Secara teknis, implementasi *sandboxing* modern sering kali memanfaatkan fitur kernel sistem operasi tingkat rendah, seperti control groups (cgroups) dan namespaces pada Linux. Pendekatan ini memungkinkan sistem penilaian untuk memantau penggunaan memori dan waktu eksekusi secara presisi, serta mematikan proses secara paksa apabila terdeteksi melanggar batasan yang telah dikonfigurasi. Penerapan teknologi *sandboxing* memberikan jaminan mutlak bahwa setiap kode tereksekusi dalam ruang yang benar-benar terisolasi, sehingga melindungi integritas sistem utama dari potensi intervensi yang berbahaya.

### 2.3.5. *Websocket*

*WebSocket* adalah protokol komunikasi komputer yang menyediakan saluran komunikasi dua arah penuh atau *full-duplex* melalui koneksi TCP tunggal. Lombardi mendefinisikan *WebSocket* sebagai teknologi yang dirancang untuk mengatasi keterbatasan model *request-response* pada HTTP, di mana setiap pertukaran data memerlukan inisiasi berulang dari sisi klien. Berbeda dengan HTTP yang bersifat *stateless* dan memiliki *overhead* tinggi akibat pengiriman header yang berulang, *WebSocket* memungkinkan koneksi tetap terbuka setelah proses *handshake* awal berhasil dilakukan. Hal ini memungkinkan server untuk mengirimkan data secara aktif ke klien kapan saja tanpa perlu menunggu permintaan, menjadikan pertukaran data le bih efisien dan responsif.

Mekanisme kerja WebSocket dimulai dengan permintaan HTTP yang menyertakan header khusus yang memberi sinyal kepada server untuk beralih protokol dari HTTP ke WebSocket. Setelah koneksi terbentuk, data dapat dikirimkan dalam bentuk *frame* dengan *overhead* minimal yang secara signifikan

mengurangi latensi jaringan dibandingkan metode tradisional seperti *polling* atau *long-polling*. Lombardi menekankan bahwa efisiensi ini sangat krusial untuk aplikasi modern yang membutuhkan interaktivitas tinggi, karena jalur komunikasi yang selalu terbuka menghilangkan keterlmabatan yang biasanya terjadi saat membangun koneksi baru untuk setiap paket data [23].

Dalam konteks pengembangan system pada pembangunan *Online Judge*, penerapan *WebSocket* memegang peranan vital untuk menjamin pengalaman pengguna yang lancar. *Websocket* memungkinkan sistem untuk memberikan umpan balik status penilaian secara *real-time* kepada pengguna tanpa perlu membebani server dengan permintaan pembaruan yang terus-menerus. Penggunaan *WebSocket* memungkinkan antarmuka pengguna menampilkan perubahan status submisi secara instan, mulai dari antrean, kompilasi, hingga hasil akhir. Mekanisme ini memastikan transparansi dan responsivitas sistem yang tinggi selama kompetisi berlangsung.

### 2.3.6. *Relational Database*

Basis data relasional adalah sistem pengelolaan data digital yang mengorganisir informasi ke dalam struktur tabel yang terdiri dari baris dan kolom. Model ini didasarkan pada teori relasional yang diperkenalkan oleh Codd, di mana integritas data dijaga melalui penggunaan komponen terstruktur seperti kunci primer, kunci asing, dan *constraints* [24]. Struktur yang terdefinisi dengan ketat ini memungkinkan sistem untuk menerapkan aturan validitas data yang spesifik, sehingga menjamin bahwa informasi yang disimpan tetap konsisten dan terhindar dari redundansi yang tidak perlu. Dalam pengembangan sistem informasi modern, pendekatan ini menjadi fondasi utama ketika akurasi hubungan antar entitas data menjadi prioritas yang lebih tinggi dibandingkan fleksibilitas skema.

Pemilihan teknologi basis data yang tepat sangat krusial karena berdampak langsung pada kinerja sistem, terutama dalam aspek kecepatan waktu respons. Sebuah penelitian menunjukkan bahwa PostgreSQL memiliki kinerja yang sangat efisien dan stabil dalam menangani kueri kompleks yang melibatkan pemrosesan data dalam jumlah besar [25]. Teknologi ini terbukti mampu menjaga efisiensi

waktu eksekusi yang optimal bahkan ketika dihadapkan pada beban kerja transaksional yang berat dan skenario konkurensi tinggi. Keunggulan performa ini menjadikan PostgreSQL pilihan yang sangat relevan untuk sistem yang menuntut pemrosesan data yang cepat dan presisi tanpa mengorbankan stabilitas operasional.

### 2.3.7. *Object Storage*

*Object storage* merupakan arsitektur penyimpanan data yang mengelola informasi sebagai objek diskrit. Setiap objek dalam arsitektur ini terdiri dari data objek, *metadata* yang mendeskripsikan karakteristik data, serta identifikasi unik yang memungkinkan akses langsung tanpa perlu mengetahui lokasi fisik penyimpanannya. Pendekatan ini menggabungkan keunggulan akses langsung seperti pada *Storage Area Network* dengan kemudahan berbagi data lintas platform seperti pada *Network-Attached Storage*. Dengan memisahkan manajemen *metadata* dari aplikasi penyimpanan, object storage memungkinkan skalabilitas yang jauh lebih besar [26].

Salah satu implementasi modern dari teknologi ini adalah MinIO, sebuah layanan penyimpanan objek berkinerja tinggi yang kompatibel dengan protokol Amazon S3. MinIO dirancang khusus untuk lingkungan *cloud-native* dan sangat efektif ketika diintegrasikan dengan teknologi kontainerisasi seperti Docker. Solusi ini menawarkan efisiensi tinggi dalam mengelola data tidak terstruktur dalam jumlah besar karena kemampuannya untuk beroperasi secara terdistribusi. Penelitian menunjukkan bahwa penggunaan MinIO dalam lingkungan berbasis kontainer mampu menjaga stabilitas kinerja sistem dalam aspek kecepatan akses data dan efisiensi penyimpanan, bahkan ketika terjadi peningkatan volume data yang signifikan [27].

### 2.3.8. Application Programming Interface (API)

*Application Programming Interface* adalah sekumpulan definisi dan protokol yang memungkinkan berbagai komponen perangkat lunak untuk berkomunikasi satu sama lain. Dalam sistem terdistribusi, API berfungsi sebagai jembatan atau lapisan abstraksi antara konsumen data dan penyedia data [28].

Penggunaan API memberikan beberapa keuntungan signifikan dibandingkan akses langsung ke sumber data. API menciptakan decoupling antara klien dan server. Kemudian, API menyederhanakan interaksi dengan menyediakan endpoint yang terdefinisi dengan baik, sehingga klien tidak perlu menulis query database yang kompleks.

Dalam arsitektur microservices, REST atau dikenal sebagai Representational State Transfer adalah gaya arsitektur yang umum digunakan untuk merancang API. REST API menggunakan protokol HTTP dan mengidentifikasi setiap sumber daya dengan *Uniform Resource Locator* yang unik. Untuk memastikan interoperabilitas dan kemudahan penggunaan, API modern sering kali didesain sesuai dengan OpenAPI Specification (OAS). Spesifikasi ini menyediakan standar agnostik terhadap bahasa pemrograman untuk mendeskripsikan API, sehingga kapabilitas layanan dapat dipahami dengan mudah baik oleh manusia maupun mesin tanpa perlu mengakses kode sumber atau dokumentasi tambahan.

### 2.3.9. Black Box Testing

Black Box Testing merupakan metode pengujian perangkat lunak yang berfokus pada validasi fungsionalitas sistem berdasarkan spesifikasi kebutuhan tanpa memeriksa struktur kode internalnya. Pendekatan ini memperlakukan perangkat lunak sebagai sebuah kotak hitam, di mana penguji hanya mengevaluasi kesesuaian antara masukan yang diberikan dan keluaran yang dihasilkan untuk menemukan kesalahan pada antarmuka maupun logika bisnis. Metode ini dinilai sangat efektif untuk memverifikasi alur kerja aplikasi dari sudut pandang pengguna akhir, memastikan bahwa setiap fitur berjalan sesuai dengan skenario operasional yang telah dirancang sebelumnya [29].

Dalam konteks pengembangan platform *coding contest*, metode ini memiliki relevansi tinggi untuk menjamin akurasi fitur-fitur vital yang berinteraksi langsung dengan peserta. Karakteristik sistem kompetisi yang sangat bergantung pada ketepatan penilaian otomatis menuntut validasi fungsional yang ketat terhadap berbagai kemungkinan masukan kode dari pengguna. Penerapan pengujian ini memastikan bahwa sistem mampu memberikan status penilaian dan pembaruan

papan peringkat yang akurat.

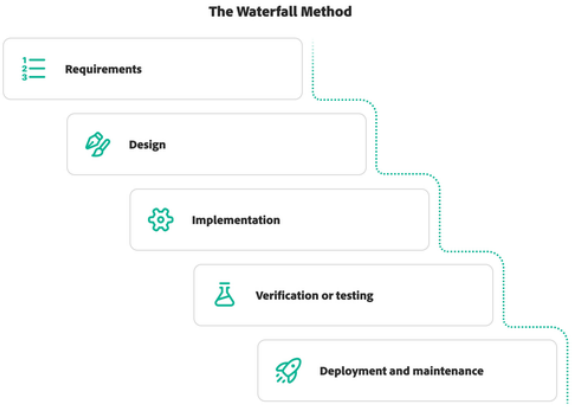
### 2.3.10. *Load Testing*

*Load testing* adalah metode pengujian kinerja yang bertujuan memvalidasi perilaku sistem saat dihadapkan pada beban kerja tertentu, seperti jumlah pengguna konkuren yang tinggi. Berbeda dengan pengujian fungsional yang hanya memeriksa kebenaran logika, pengujian ini memastikan sistem memenuhi persyaratan non-fungsional vital seperti waktu respons dan throughput. Jiang dan Hassan menekankan bahwa pengujian ini sangat krusial bagi sistem berskala besar untuk mendeteksi masalah skalabilitas yang sering kali tidak terlihat selama fase pengembangan awal. Tanpa pelaksanaan pengujian ini, risiko kegagalan sistem atau *downtime* saat menghadapi lonjakan lalu lintas data yang sebenarnya menjadi sangat tinggi [30]. Manfaat utama dari pelaksanaan load testing adalah memberikan pembuktian empiris mengenai batas maksimal latensi dan throughput, sehingga potensi kegagalan sistem akibat lonjakan trafik ekstrem dapat dideteksi dan dimitigasi lebih dini.

### 2.3.11. Waterfall

Model Waterfall yang dikenal sebagai model siklus hidup linear sekuensial merupakan metode pengembangan perangkat lunak yang paling mendasar dan terorganisir. Menurut Shylesh, pendekatan ini mengharuskan penyelesaian penuh pada setiap fase sebelum dapat melangkah ke fase berikutnya yang berarti tidak ada aktivitas yang berjalan tumpang tindih di antara tahapan tersebut. Alur kerjanya bergerak secara sistematis dari analisis kebutuhan, perancangan, implementasi, pengujian, hingga pemeliharaan layaknya aliran air terjun. Metode ini sangat mengutamakan perencanaan dan dokumentasi yang matang di awal proyek karena hasil keluaran dari satu tahap berfungsi sebagai dasar mutlak bagi tahap selanjutnya. Kedisiplinan dan struktur yang ketat ini menjadikan model Waterfall sangat relevan untuk diterapkan dalam penelitian ini karena mampu menjamin seluruh proses pengerjaan dari awal hingga akhir dapat berjalan secara terukur, memiliki batasan ruang lingkup yang jelas, serta menghasilkan dokumentasi yang lengkap dan rapi

[31].




**Gambar 2. 2 Tahapan Model Waterfall**

### 2.3.12. Spring Boot

Spring Boot adalah sebuah *framework* berbasis Java yang dirancang untuk menyederhanakan proses pengembangan aplikasi *backend*, khususnya yang mengadopsi arsitektur microservices [32]. Framework ini dibangun di atas platform Spring dengan pendekatan "*convention-over-configuration*" yang secara drastis mengurangi jumlah kode *boilerplate* dan konfigurasi manual yang diperlukan oleh pengembang. Dengan menyediakan konfigurasi default yang komprehensif dan server aplikasi yang embedded, Spring Boot memungkinkan pengembang untuk membuat aplikasi backend tingkat produksi yang dapat langsung dijalankan dengan cepat. Kemampuannya untuk memberdayakan pembuatan dan deployment layanan secara independen menjadikannya pilihan utama untuk paradigma aplikasi bisnis yang ringan dan modular. Arsitektur aplikasi Spring Boot umumnya mengikuti pola berlapis untuk memastikan *separation of concerns* yang terdefinisi dengan jelas. Lapisan tersebut terdiri dari 3 lapisan utama yaitu *Controller*, *Service*, dan *Repository*.

### 2.3.13. React.js

React JS adalah sebuah library JavaScript *open-source* dikembangkan oleh Facebook, yang secara khusus berfokus pada pembangunan antarmuka pengguna *User Interface* untuk aplikasi web *Single Page Application* [33]. Prinsip utama dari

React JS adalah menerapkan arsitektur berbasis komponen, di mana User Interface dipecah menjadi bagian-bagian kecil yang independen dan dapat digunakan kembali sebagai *reusable components*. Setiap komponen mengelola *state* miliknya sendiri dan ketika *state* tersebut berubah, React akan secara efisien memperbarui dan *render* ulang komponen. Hal ini menyederhanakan proses pengembangan *User Interface* yang kompleks dan memungkinkan pengembang untuk lebih fokus pada logika bisnis aplikasi.

Salah satu inovasi utama yang membuat React.js sangat efektif dan efisien adalah penggunaan Virtual DOM. Dibandingkan dengan memanipulasi DOM aplikasi web secara langsung, React.js menyimpan representasi DOM dalam memori. Sehingga jika terjadi perubahan state, React.js terlebih dahulu memperbarui Virtual DOM dan membandingkan versi Virtual DOM sebelum dan sesudah perubahan. Selanjutnya, React.js hanya akan menerapkan perubahan yang benar-benar diperlukan pada DOM aplikasi web yang asli, sehingga meminimalkan operasi yang menggunakan proses kompleks dan meningkatkan kecepatan render secara signifikan. Pendekatan alur data satu arah membuat status aplikasi lebih prediktif, menjadikan React.js sebagai pilihan yang populer dan kuat untuk membangun antarmuka web yang modern, cepat, dan interaktif.