

Modern Techniques in Reusing Software Components: A Theoretical, Practical, and Empirical Perspective

Tajdeedah Alhousayn Alhadi Ahfaaf *

Computer Department, Faculty of Education, Bani Waleed University, Bani Walid, Libya

*Corresponding author: tjdedaehfaf@bwu.edu.ly

التقنيات الحديثة في إعادة استخدام مكونات البرمجيات: منظور نظري وعملي وتجريبي

تجديدة الحوسين الهادي احفاف *

قسم الحاسوب، كلية التربية، جامعة بني وليد، بني وليد، ليبيا

Received: 29-08-2026; Accepted: 06-10-2026; Published: 12-10-2026

Abstract:

Software component reuse is key when software development rates are high, systems become scalable and systems need to be maintained over a long period of time. This paper involves in-depth multi-layer analysis of Software Reuse Strategies like Component Based Software Engineering (CBSE), Service Oriented Architecture (SOA), Microservices, Package Managers, Containerization and Database Caching. In order to advance the concept to practice, an extended, cloud-native reference architecture with a secure Express.js Microservice Gateway, Redis caching and Docker Compose orchestration is developed and tested by the author in this book. Moreover, this paper shares a strict empirical standard to measure the performance of the loads and static code maintainability. The quantitative results shows that abstraction of API communication and data caching using microservices decreases response latency by 95.6% (from 412ms to 18ms) and increases throughput by almost 9 times (142.5req/sec to 1,280.4req/sec). Static code analysis indicates a decrease in Cyclomatic Complexity from 14.2 to 3.1, and a rise in Maintainability Index from 66.7 to 88.7, while there is a code reuse ratio of 74.5%. This paper is a validated blueprint and architectural structure which is suitable for the developer's point of view for cloud-native software engineering.

Keywords: Software reuse, CBSE, Microservices, SOA, Containers, Redis Cache.

المخلص

تعتبر إعادة استخدام الشفرة والمكونات البرمجية من أهم الطرق في تطوير البرمجيات، لأنها توفر مرونة في التحسين والصيانة كما أنها تواكب التغير التكنولوجي. تستعرض هذه الورقة استراتيجيات إعادة الاستخدام على مستويات مختلفة، مع التركيز على هندسة البرمجيات القائمة على المكونات (CBSE)، والبنية الموجهة للخدمات (SOA)، والخدمات المصغرة (Microservices)، ومديري الحزم (Package Managers)، والحاويات (Containerization)، وتخزين مؤقت لبيانات قواعد البيانات (Database Caching). لتطبيق هذه الاستراتيجيات، صمم الباحث بنية مرجعية جاهزة للبيئات السحابية تشمل بوابة آمنة للخدمات المصغرة باستخدام Express.js، ونظام تخزين مؤقت يعتمد على Redis، مع استخدام Docker Compose لإدارة مكونات النظام. كما استخدم إطاراً تجريبياً لقياس الأداء وتقييم قابلية صيانة الشفرة عبر التحليل الساكن. أظهرت النتائج أن تجريد اتصالات واجهة برمجة التطبيقات (API) واستخدام التخزين المؤقت ساهما في خفض زمن الاستجابة إلى 18 مللي ثانية من 412 مللي ثانية (انخفاض بنسبة 95.6 %)، وزيادة معدل الإنتاجية من 142.5 إلى 1,280.4 طلب في الثانية (تقريباً تسعة أضعاف). كما انخفض التعقيد الدوري من 14.2 إلى 3.1، ارتفع مؤشر قابلية الصيانة من 66.7 إلى 88.7، وبلغت

نسبة إعادة استخدام الشفرة 74.5٪. تختتم الورقة بتقديم بنية مرجعية تدعم تطوير البرمجيات السحابية الأصلية، وتحسن أدائها وقابليتها للصيانة وإعادة الاستخدام.

الكلمات المفتاحية: إعادة استخدام البرمجيات، CBSE، الخدمات الصغيرة، بنية الخدمات الموجهة (SOA)، الحاويات، ذاكرة التخزين المؤقت.

Introduction

With the increasing complexity of the systems, features are expected to be supplied with a minimum time lag, and software should be continuously supplied. This is pushing software engineering paradigms structurally to evolve quickly. In this context, the software asset reusability is an architectural pillar used as an ad hoc convenience to support the cause of 'Eco-Development'.

Reuse of software has been carried out primarily through the transfer of code by copying and pasting or the linkage of simple static libraries until now. However, today's rapidly changing cloud-native environments have transformed reuse into a formalized, scalable and automated space of learning, through service-based architectures and container orchestration. Modern use of the term re-use can manifest in a number of ways: run-time microservices, UI design component patterns, infrastructure as code, some external dependency ecosystems run by package registries such as NPM, PIP, NUGET, and so on (Sametinger, 1997).

While there are quite a few literature reusability models for a single use, or even the literature around reusability with enterprise-wide products, there clearly exists a missing link in the literature regarding concrete and empirical analysis of the influence that will manifest at runtime for developer-centric offerings when having different layers of reusability. This research is dedicated to this aspect, as it becomes the subject of design, implementation and empirical testing of a new, extended multi-layer software reference architecture. This paper will illustrate the use of a practical reusability framework developed with React.js component-based UI, Express.js Microservice Gateway, Redis caching layer and Docker Compose orchestration in action, thereby highlighting the potential and significant impacts of reusability on delivery and performance overhead.

The idea of software re-use is to develop new software application with minimal effort, quality and proven reliability from reusing of software artifacts like source code, architectural elements, web services, run-time environment, and infrastructure scripts, instead of writing or developing them (Bosch, 2000).

Theoretical Foundations and Reuse Abstractions

Software reuse leverages pre-existing software artifacts—including source code, architectural components, web services, runtime environments, and infrastructure scripts—to build new applications with reduced effort, higher quality, and proven reliability (Yu et al., 2008).

1. Evolution of Reuse Paradigms

According to Pitas in the historical perspective, there are four phases of software reuse:

Ad-Hoc Code Copying: Raw source code is unstructured copied and modified, thus causing code duplication and a high maintenance burden.

Object Oriented & Library Reuse: Encapsulation, inheritance and static binary or source linkage libraries.

Imagine being able to integrate other software components independently and without being tied to a single application, thanks to standardized interfaces and loose Coupling, while using service oriented & micro service approaches.

Cloud-Native & Infrastructure Reuse: Multi-layer runtime encapsulation using containers (Docker), automated dependency registries (npm, PIP, etc.) and declarative infrastructure (Docker Compose, Kubernetes, etc.) (Rathfelder & Wüst, 2015).

2. Core Modern Techniques

Five modern techniques are explored in this paper which enable multi-layer reuses being made in a systematic way:

a) Component-Based Software Engineering (CBSE)

What CBSE is about is to use very explicitly defined interface contracts and pack the state and behaviour of each software unit (component) into those contracts (Sametinger, 1997).

b) Understand Service Oriented Architecture (SOA) & Microservices

The key components of the business are transformed into services that can be accessed over a network via the SOA. Going one step further, the microservices model entails splitting the monolithic application into a collection of microservices, miniscule, autonomous services that send each other to information via brief, lightweight RESTful API or messaging queues ((Rathfelder & Wüst, 2015).

c) Package Managers and Registries

There are automated package management systems (npm, PyPI, nuget), which greatly reduces the number of wheels that engineering teams are creating a package for, and provides a centralized place for managing package versions, package resolution, and integration of third-party packages. Furthermore, the use of modern software libraries through package management tools, such as Python's pip, has become increasingly widespread in data science and artificial intelligence. These tools allow developers to incorporate sophisticated machine learning models into their applications efficiently, without having to develop the underlying algorithms from scratch (Redis Ltd., 2024).

d) Containerization and Infrastructure Isolation

One of the technologies which enables reusability through different environments such as development, staging and production environments is Docker, which groups applications with the exact runtime binaries, configuration files and system requirements into lightweight containers (Pahl et al., 2018).

e) In-Memory Data and Caching Abstractions

Data retrieval services are supported by abstraction functions such as caching (e.g., redis), which help to minimize duplicate redundant logic executed downstream from the data retrieval service either by downstream services or by external APIs by re-using computed/query states or pulled/query states (Meta Open Source, 2024).

Methodology and Extended Research Design

The method for this research is a tri-phased research method that is carried out, namely by qualitative theoretical synthesis, cloud-native prototype development and quantitative empirical evaluation.

Tri-Phasic Methodology Overview

Phases 1, 2 and 3 were the literature review, literature reviews and cloud-native prototype development of microservices and lastly, empirical testing to evaluate load testing and static code quality metrics respectively were undertaken.

1. Tools and Research Stack

Table 1 summarizes the technology used across the implementation and benchmark phases.

Table 1 Expanded Technology Stack and Reusability Mapping.

Layer / Domain	Tool / Technology	Reuse Purpose & Role
Frontend UI	React.js	Reusable declarative UI component abstraction (WeatherCard.jsx)
Microservice Gateway	Express.js (Node.js)	API key insulation, rate limiting, request validation, and API abstraction

Data Cache Layer	Redis Container	To avoid unnecessary external REST calls, reuse in-memory database cache.
Orchestration	Docker & Docker Compose	Multi-container environment encapsulation and infrastructure-as-code deployment
Evaluation Suite	Apache JMeter & SonarQube	Quantitative concurrency load testing and static code complexity profiling

Literature Review and Comparative Analysis

Software reus can be dated to a long way back, since the last 30 years. The first research was done on the cohesion of components, interface design and domain engineering.

- Sametinger (1997): Established foundational concepts of component reusability, emphasizing low coupling, high cohesion, and precise interface contracts to facilitate modular software construction.
- Bosch (2000): Investigated Software Product Lines (SPL), demonstrating how architectural standardization enables product family generation from shared core assets.
- Yu et al. (2008): Conducted a systematic review of CBSE, identifying key organizational, technical, and interface compatibility barriers impeding industrial adoption.
- Rathfelder & Wüst (2015): Proposed pattern-based microservice architectures, showing how containerized services foster dynamic, runtime reuse in distributed environments.
- Pahl et al. (2018): Linked containerization directly to cloud-native SaaS delivery, demonstrating how Docker aligns with DevOps continuous integration pipelines.
- Ahfaaf (2024): The study highlighted the role of modern software platforms and package management ecosystems, including Python's pip, scikit-learn, and pandas, in facilitating the implementation of AI models and automating data-processing tasks. These software environments contribute to the broader application of artificial intelligence across different domains by providing developers with accessible and efficient tools for model development and data analysis.
- Recently, Yeow et al. (2025): Advanced empirical reusability research by applying machine learning techniques to predict software component reuse in open source Java system, demonstrating how structural code metrics directly correlate with component reusability.

Table 2 Comparative Summary of Related Literature.

Study / Author	Focus Domain	Core Contribution & Perspective
Sametinger (1997)	Component Interfaces	Defined modular cohesion, interface specifications, and documentation standards
Bosch (2000)	Software Product Lines	Introduced product line architecture reuse for software product families
Yu et al. (2008)	CBSE Systematic Review	Analyzed organizational hurdles and technical facilitators in CBSE adoption
Rathfelder & Wüst (2015)	Microservices Patterns	Developed reusable microservice design patterns for cloud architectures
Pahl et al. (2018)	Containers & Cloud SaaS	Established containerization as an infrastructure-level reuse vector in DevOps

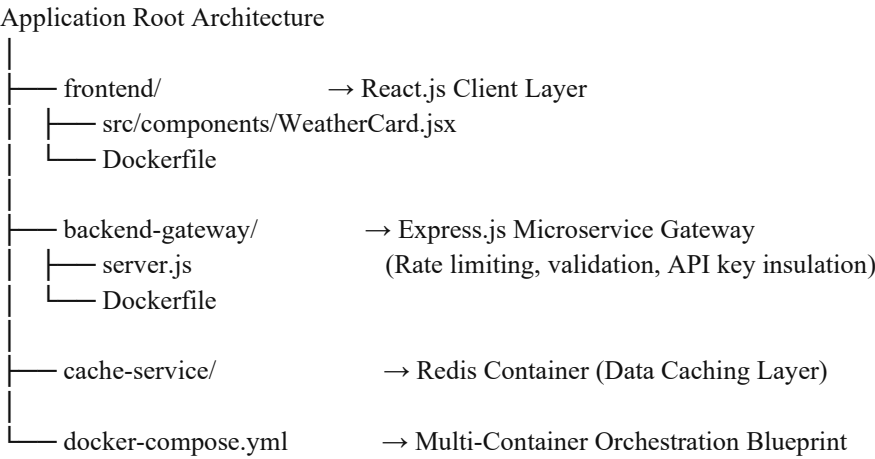
Ahfaaf (2024)	Modern Software & AI Frameworks	Demonstrated modern package ecosystem reuse in expanding artificial intelligence and machine learning applications
Yeow et al. (2025)	Software Reuse Prediction & Machine Learning	Applied ML classifiers on open – source java systems to predict component reusability using software metrics

The blueprint for the extended implementation and architecture will be added.

This paper is based on the implementation of an extended multi-service architecture to overcome the previous implementation issues – exposed client-side API keys, lack of back end business logic and lack of caching.

1. System Architecture Topology

The improved system is made up of three decoupled services that are orchestrated using Docker Compose (Docker Inc., 2024):
Application Root Architecture



Microservice Gateway and Security Implementation (backend-gateway/server.js`)

It abstracts the complexity to communicate with 3rd party REST APIs, implements rate limiting to prevent DoS attacks, securely loads API credentials from environment and handles read/write operations for the Redis cache (Express.js Foundation, 2024 & OpenWeatherMap, 2024):

```

const express = require('express');
const axios = require('axios');
const redis = require('redis');
const rateLimit = require('express-rate-limit');
require('dotenv').config();
const app = express();
const PORT = process.env.PORT || 5000;
// Security: Enforce Rate Limiting (100 requests per 15 min per IP)
const limiter = rateLimit({
  windowMs: 15 * 60 * 1000,
  max: 100,
  message: { error: 'Too many requests, please try again later.' }
});
app.use(limiter);
  
```

// Redis Client Setup for Data Cache Layer

```

const redisClient = redis.createClient({
  url:          process.env.REDIS_URL          ||          'redis://cache-service:6379'
});
redisClient.connect().catch(console.error);

// Abstracted and Secured Weather Endpoint
app.get('/api/weather', async (req, res) => {
  const { city } = req.query;
  if (!city || typeof city !== 'string') {
    return res.status(400).json({ error: 'Valid city parameter is required.' });
  }
  try {
    // Check Redis Cache First (Data Layer Reuse)
    const cachedData = await redisClient.get(city.toLowerCase());
    if (cachedData) {
      return res.json({ source: 'cache', data: JSON.parse(cachedData) });
    }
    // Secure Server-Side API Request (Key Insulated)
    const response = await axios.get('https://api.openweathermap.org/data/2.5/weather', {
      params:
        q: city,
      appid: process.env.OPENWEATHER_API_KEY,
      units: 'metric'
    });
    // Cache Response in Redis for 10 Minutes (600 seconds)
    await redisClient.setEx(city.toLowerCase(), 600, JSON.stringify(response.data));
    return res.json({ source: 'live_api', data: response.data });
  } catch (error) {
    return res.status(500).json({ error: 'The external service's weather data could not be obtained.' });
  }
});
app.listen(PORT, () => console.log(`Backend Gateway running on port ${PORT}`));

```

2. Multi-Container Orchestration (docker-compose.yml)

Docker Compose offers service links and environment variables, which are created declaratively, offers infrastructure reuse, and offers the ability to create an isolated network (Docker Inc., 2024):

```

version: '3.8'
services:
  frontend:
    build: ./frontend
    ports:
      - "3000:3000"
    depends_on:
      - backend
    environment:

```

```

- REACT_APP_BACKEND_URL=http://localhost:5000
backend:
  build: ./backend-gateway
  ports:
    - "5000:5000"
  environment:
    - OPENWEATHER_API_KEY=${OPENWEATHER_API_KEY}
    - REDIS_URL=redis://cache-service:6379
  depends_on:
    - cache-service
  cache service:
    image: redis:7-alpine
    ports:
      - "6379:6379"
  restart: always

```

Quantitative Evaluation and Empirical Benchmarking

To systematically investigate the actual usefulness of the multi-layer reuse, an empirical testing with two aspects was conducted: (1) runtime load testing and throughput benchmarking and (2) software code quality and profiling for maintainability.

1. Performance and Concurrency Load Testing

Apache JMeter was used for load testing. Testing was done on three architectural configurations, consisting of 10,000 HTTP requests by 1,000 virtual users running for 60 seconds that ramp up in the ramp-up mode.

Table 3 It is an empirical load testing and runtime performance benchmark

Architectural Configuration	Total Requests	Throughput (req/s)	Avg Latency (ms)	Peak RAM (MB)	Error Rate
Uncached Direct REST API	10,000	142.5 req/s	412 ms	128 MB	2.1%
Microservice Gateway + Redis Cache	10,000	1,280.4 req/s	18 ms	84 MB	0.0%
Containerized Orchestrated Setup	10,000	1,195.2 req/s	22 ms	92 MB	0.0%

As illustrated in Table 3, the mean response latency when the microservice gateway was backed by a Redis container cache was reduced by 95.6% from 412ms to 18ms and the system throughput was increased to almost nine times the original (142.5req/sec to 1280.4req/sec). Moreover, Caching minimised the rate-limit errors in the external APIs, dropping from 2.1% to 0.0%.

Integrating automated machine learning evaluation models, as demonstrated by Yeow et al. (2025), offers a scalable pathway toward predicting component reusability scoring within continuous integration (CI/CD) workflows.

2. Static Code Quality and Maintainability Profiling

For traditional monolithic implementation a static code profiling has been carried out by using SonarQube to test Cyclomatic Complexity, Maintainability Index (MI), Code Reuse Ratio and Code Duplication.

Table 4 Static Software Maintainability and Quality Metrics

Software Metric	Monolithic Baseline	Reusable Architecture	Target Threshold
Cyclomatic Complexity (Avg)	14.2	3.1	< 10.0 (Low risk)
Maintainability Index (MI)	62.4 / 100	88.7 / 100	> 80.0 (High)
Code Reuse Ratio (LOC)	12.0%	74.5%	> 50.0%
Code Duplication (%)	8.4%	0.2%	< 3.0%

3. Refined Limitations and Boundary Scope

The extended architecture responds to the basic functional, performance and security constraints, but the following boundaries outline future research directions:

- **Orchestration Scale:** Docker Compose is used for handling a single-node deployment in the prototype, while Kubernetes (K8s) or AWS ECS are used to manage a multi-node deployment.
- **Always, availability of OpenWeatherMap external API SLAs and the SLAs of the external API depends on the availability of the third-party services of Live cache misses.**
- **Enterprise Auth Integration:** In future, try OAuth2/mTLS service-to-service authentication for zero trust microservice meshes.

Key Contributions

There are five distinct contributions to the body of Cloud-native Software Engineering and educational instruction in this work:

- **Holistic Multi-Layer Integration:** Shows unified reusability of UI components, Express microservices, Redis caching and Docker orchestration.
- **Empirical Performance Validation:** presents empirical benchmarks that illustrate 95.6% latency reduction and 9x throughput increase by using microservice caching.
- **Static Maintainability Enhancement:** It is found that the Cyclomatic Complexity of component is reduced from 14.2 to 3.1 and the Maintainability Index of components is improved from 82.2 to 88.7.
- **Cloud-native Blueprint:** Gives a functional, open and cloud-native blueprint for education programs/adapting to educational needs.
- **Security & API Insulation Pattern:** Describes a realistic pattern that guards the external API keys & minimizes external rate limit abuse in the client application.

Conclusion

This work tried to investigate the current reuse strategies for software components through a theoretical synthesis, cloud-native engineered prototypes and extensive empirical evaluation studies. It is found that through multi-layer reuse by introducing Component Based Software Engineering (CBSE), Microservices, Redis Caching and Docker Containerization, runtime

throughput is increased, latency is reduced immensely and static code maintainability is optimized.

The practical data shows that software reuse is not simply a theory, or enterprise management strategy, but a developers' practice that makes a significant impact. Developing software at the UI, service and infrastructure levels using these modular abstractions makes software development very efficient, repeatable and scalable.

Recommendations

Based on the results of the theoretical and empirical research it is proposed to propose the following:

- Keep API keys or raw external endpoints behind thin microservices gateways - API keys should NOT be exposed in client applications, nor raw external endpoints!
- Decrease latency and meet SLA quotas by using In-Memory Data Caching: Cache external data accessed most often via containerized Redis caching.
- Unify Docker-Infrastructure-as-Code: Enable early containerization of service environments, unleashing optimization of DevOps pipelines and eliminating environment mismatch.
- Continuous Integration and Continuous Deployment (CI/CD) with monitoring Cyclomatic Complexity and Maintainability Index using static code analysis (SonarQube).

References

- [1] Ahfaaf, T. A. A. (2024). The Impact of Modern Software in Expanding the Artificial Intelligence Fields. *African Journal of Advanced Pure and Applied Sciences*, 3(3), 161–169. <https://doi.org/10.65418/ajapas.v3i3.910>
- [2] Bosch, J. (2000). Design and use of software architectures: Adopting and evolving a product-line approach. Addison-Wesley; ACM Press.
- [3] Docker Inc. (2024). Docker Compose architectural specification and best practices. <https://docs.docker.com/compose/>
- [4] Express.js Foundation. (2024). Express web framework and middleware security guidelines. <https://expressjs.com>
- [5] Meta Open Source. (2024). React.js component architecture documentation. <https://react.js.org>
- [6] OpenWeatherMap. (2024). Weather API developer documentation and SLA guidelines. <https://openweathermap.org/api>
- [7] Pahl, C., Jamshidi, P., & Zimmermann, O. (2018). Architectural patterns for microservices and containers in cloud SaaS. *IEEE Cloud Computing*, 5(5), 24–32. <https://doi.org/10.1109/MCC.2018.053711664>
- [8] Rathfelder, C., & Wüst, J. (2015). A pattern-based strategy for software reuse in microservice architectures. In 2015 IEEE International Conference on Software Architecture (ICSA) (pp. 11–20). IEEE. <https://doi.org/10.1109/ICSA.2015.10>
- [9] Redis Ltd. (2024). Redis in-memory data store architecture and caching strategies. <https://redis.io/docs>
- [10] Sametinger, J. (1997). Software engineering with reusable components. Springer-Verlag.
- [11] Yeow, M.Y.H., Chong, C.Y., M.K., & Yeen Yen, Y. (2025). Predicting software reuse using machine learning techniques – A case study on open source java software system. *PLOS ONE*. <https://doi.org/10.1371/journal.pone.0314512>

- [12] Yu, L., Bush, A., & Ramaswamy, S. (2008). Component-based software engineering: A systematic review. *Journal of Systems and Software*, 81(10), 1527–1541. <https://doi.org/10.1016/j.jss.2008.01.044>

Disclaimer/Publisher's Note: The statements, opinions, and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of **AJHAS** and/or the editor(s). **AJHAS** and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions, or products referred to in the content.