# CHAPTER 5
# IMPLEMENTATION AND RESULTS

## 5.1. Implementation

This project uses the Go programming language. In this project, the performance of monolith architecture and microservice architecture is compared by load testing both architecture. Both architectures contain the same amount of services. Those services are built as REST API services where services communicate with the outside world using JSON format. In this project, both architectures are load-tested using JMeter. Each service has its own unique features that make it different from one another.

In monolith, all service connect to single database. This project use PostgreSQL as the database. In order to minimalize latency, every runtime service call database only once so that there won't be too much connection.

```
1. func DBCon() *sql.DB {
2.       db,  err  :=  sql.Open("postgres",  "host=127.0.0.1  port=5433
   user=postgres password=admin dbname=dbmerch sslmode=disable")
3.     if err != nil {
4.         log.Fatal(err)
5.     }
6.     return db
7. }
```

This function is called just once in the main package. On line 2, sql.Open function used for making a connection to dbmerch. DBCon function's sole purpose is to return database connection in form of a pointer variable with the type of SQL as seen on line 8. If facing an error, line 4 in the function will stop the program and return an error message on the terminal.

In Go, like in Java programming, the executable functions are all located in the main function. Since connecting to the database should be secured and not exposed to the outside world. In order to achieve that, all functions that connect to the database are separated into a different package. Since there are 3 main services, there are 3 repositories. Package repository handles all functions responsible to make queries.

Merchant service contain basic CRUD function. That means repository has to contain create function, update function, get function, delete function, and also get function.

```
8. func InsertMerchant(request model.Merchant, db *sql.DB) (model.Merchant,
   error) {
9.     var result model.Merchant
10.     query := `INSERT INTO tblmerchant (name)
11.     VALUES ($1)
12.     RETURNING idmerchant`
13.     err := db.QueryRow(query, request.Name).
14.         Scan(&result.Idmerchant)
15.     if err != nil {
16.         fmt.Println("Error Repo ", err)
17.         return result, err
18.     }
19.     return result, nil
20. }
```

InsertMerchant function duty is to insert data to the table like explained in line 10. This function will return struct merchant and error type variable. The difference between repository functions located on the query written. This means the Get function's query is to select data from the table, the update function's query is to update the data, and the delete function is to remove data from the table.

The key feature of user service is the login feature. Login feature only need to communicate with the database when getting the data for checking user identity.

```
21. func GetUserByEmail(email string, db *sql.DB) (model.User, error) {
22.     var result model.User
23.     err := db.QueryRow("SELECT id, email, password FROM tbluser where
   email = $1", email).Scan(&result.Id, &result.Email, &result.Password)
24.     if err != nil {
25.         fmt.Println("Error Repo ", err)
26.         return result, err
27.     }
28.     return result, nil
29.
30. }
```

Function GetUserByEmail is used to get user identity by searching the data by email. This function returns the user struct model. On line 23, QueryRow is used because the query only executed once and expected one struct as a return. Not only login feature, but user service also have register feature, like merchant insert feature, the register repository function is similar.

The transaction service key feature is bulk insert. Based on real-life transaction activity, one transaction usually contains more than one item. The repository function goal is to insert many data at once.

```
31. func        InsertTransactionDetail(request[]model.TransactionDetail,db
   *sql.DB) (bool, error) {
32.
33.     vals := []interface{}{}
34.             query     :=     `INSERT     INTO     tbltransactiondetail
   (transaction_code,product_code,quantity,price)
35.     VALUES`
36.     for _, row := range request {
37.         query += "(?, ?, ?, ?),"
38.             vals  =  append(vals,  row.TransactionCode,  row.ProductCode,
   row.Quantity, row.Price)
39.     }
40.     query = query[0 : len(query)-1]
41.     query = ReplaceSQL(query, "?")
42.     stmt, _ := db.Prepare(query)
43.     _, err := stmt.Exec(vals...)
44.
45.     if err != nil {
46.         fmt.Println("Error Repo ", err)
47.         return false, err
48.     }
49.
50.     return true, nil
51.
52. }
```

To bulk insert the data, the query could be written like line number 34. Later the query variable string concat by using a loop. Since bulk insert length depends on the input, the query is first written with “?” first. The loop is used to replace “?” with the “$” which represents the variable order. To prevent query injection, Go allow query executed with dollar sign variable. The query variable when executed changes the “$” variable with the variable from the parameter.  All of this process is done in order to make the query insert is followed with the values needed.

Like in many other programming languages, to simplify variable writing, a struct is used. All struct that used global is declared on package model.

```
53. package model
54.
55. type Merchant struct {
56.     Idmerchant int    `json:"idmerchant"`
57.     Name       string `json:"name"`
58. }
59. type User struct {
60.     Id       int    `json:"id"`
61.     Email    string `json:"email"`
62.     Password string `json:"password"`
63. }
64. type Transaction struct {
65.     Id              int    `json:"id"`
66.     TransactionCode string `json:"transactionCode"`
67.     TransactionDate string `json:"transactionDate"`
68.     PaymentMethod   string `json:"payment"`
69.     Total           int    `json:"total"`
70. }
71. type TransactionDetail struct {
72.     Id              int    `json:"id"`
73.     TransactionCode string `json:"transactionCode"`
74.     ProductCode     string `json:"productCode"`
75.     Quantity        int    `json:"quantity"`
76.     Price           int    `json:"price"`
77. }
```

All the struct declared above contains more than one variable. Those variables have the 'JSON' attribute then followed with string. Those strings are used for JSON struct declare. These structs are the blood of the application, every function passed and receives these structs from other functions or even packages.

Usecase package is where the business logic lives. Every service's business logic is coded in the usecase package. Usecase package is also responsible to call the repository function. Below is the code of merchant service usecase function.

```
78. func InsertMerchant(request model.Merchant, db *sql.DB)
   model.ResponseMerchant {
79.     var result model.ResponseMerchant
80.     merchant, err := repo.InsertMerchant(request, db)
81.     if err != nil {
82.         fmt.Println("Err Usecase", err)
83.         result.Status = 401
84.         result.Desription = err.Error()
85.         return result
86.     }
87.     merchant.Name = request.Name
88.     result.Status = 200
89.     result.Desription = "Berhasil"
90.     result.Merchant = merchant
91.     return result
92. }
```

On line 87, the InsertMerchant function calls the repository function and sends the parameter. For basic CRUD, use-case functions do not have many responsibilities except passing the data to the endpoint and triggering the repository function.

Since the user service has login features, the use-case function for this service is more complex than the merchant's use-case function. Below is the function of login service.

```
93. func LoginUser(request model.RequestLogin, db *sql.DB)
   model.ResponseLogin {
94.     var result model.ResponseLogin
95.
96.     users, err := repo.GetUserByEmail(request.Email, db)
97.     if err != nil {
98.         fmt.Println("Err Usecase", err)
99.         result.Status = 401
100.        result.Desription = err.Error()
101.        return result
102.    }
103.
104.    x := CheckPasswordHash(request.Password, users.Password)
105.
106.    if !x {
107.        fmt.Println("Wrong Password", err)
108.        result.Status = 401
109.        result.Desription = "Wrong Password"
110.        return result
111.    }
112.
113.    token, _ := GenerateToken(users.Id, users.Email)
114.    result.Status = 200
115.    result.Desription = "Berhasil"
116.    result.Email = users.Email
117.    result.Id = users.Id
118.    result.Token = token
119.
120.    return result
121.
122.}
```

The function above receives email and password as a parameter and returns the response login struct which contains email, id, and JWT. Since login in REST API application is to generate JWT, JWT is used as authorization token while accessing private API which needs authorization and secured access. Line 96 is when the function triggers the repository. The password which returned was checked on line 104 to check whether the input password is correct. If the password is correct, line 113 will trigger the GenerateToken function to make the JWT. Later in lines 114 until 118 all the data processed was inserted inside the result struct which later returned on line 120.

For transaction service, the function can call more than one repository function since the function require to insert two table at once. Below is the function code.

```
123.func  InsertTransaction(request  model.RequestTransaction,  db  *sql.DB)
   model.ResponseAddTransaction {
124.    var result model.ResponseAddTransaction
125.    var trans model.Transaction
126.    var count int
127.    times := Timestamp()
128.    count = 0
129.
130.    timesufix := time.Now().Format("20060102150405")
131.    code := "TRX-" + timesufix + "-"
132.
133.    trans.TransactionCode = code
134.     trans.TransactionDate = times
135.     trans.Total = count
136.     trans.PaymentMethod = request.PaymentMethod
137.
138.     y, err := repo.InsertTransaction(trans, db)
139.     if err != nil {
140.         fmt.Println("Err Usecase", err)
141.         result.Status = 401
142.         result.Desription = err.Error()
143.         result.Success = false
144.         return result
145.     }
146.
147.     for i, x := range request.TransactionDetail {
148.         count = count + x.Price
149.         request.TransactionDetail[i].TransactionCode = y
150.     }
151.
152.    _, err = repo.InsertTransactionDetail(request.TransactionDetail, db)
153.     if err != nil {
154.         fmt.Println("Err Usecase", err)
155.         result.Status = 401
156.         result.Desription = err.Error()
157.         result.Success = false
158.         return result
159.     }
160.
161.     result.Status = 200
162.     result.Desription = "Insert Success"
163.     result.Success = true
164.
165.     return result
166.
167. }
```

This function is responsible to build the expected struct which is later sent to the database to be inserted on line 138. In order to make a unique transaction code, on line 131, the transaction code is given a time prefix. On line 147, the loop is used to count the price of the items purchased. Later the data is passed to the repository function. If there is no error, the result variable will be returned.

This project is building a monolith REST API web application. In order to achieve that, the server needs to have a service port accessible from the outside world. Echo framework used to listen and serve HTTP requests while providing API. Code below is a snippet of the main package where the server coded

```
168. func main() {
169.     db := config.DBCon()
170.     e := echo.New()
171.     e.POST("/merchant/add", func(c echo.Context) error {
172.         u := new(model.Merchant)
173.         if err := c.Bind(u); err != nil {
```

```
174.            return err
175.        }
176.        response := usecase.InsertMerchant(*u, db)
177.        if response.Status != 200 {
178.            return c.JSON(http.StatusOK, response)
179.        } else {
180.            return c.JSON(http.StatusOK, response)
181.        }
182. e.Logger.Fatal(e.Start(":10000"))
183.
184.    })
```

In golang, the DB connection only needs to be called once in main as shown on line 169, which later the DB passed to use case function. The main function other duties are to call the use case to complete the service. On line 170, an echo server is created and on line 171 an endpoint is created which receives POST method format requests. The binding process on line173 is done in order to save the data from the request sent. The port where all endpoint run is coded on line 182.

As mentioned before, both architectures have identical service and endpoint but the code was very different since built upon quite opposite architecture. Microservice architecture is more complex. Every service has its own database connection and has its own port. In this project, the Go Kit toolkit is used to build a microservice web application. As the go kit principles, service is divided into 4 layers while coded in one package. Since the functionality is quite similar, this section will discuss how the architecture works.

In go-kit, every logic function needs to be listed on the interface struct. This method is done because this interface struct will be the bridge for functions to communicate. Below is an example of a merchant service struct.

```
185.type Service interface {
186.    InsertMerchant(ctx context.Context, merch Merchant) (string, error)
187.    GetMerchantById(ctx context.Context, id int) (Merchant, error)
188.    GetAllMerchant(ctx context.Context) ([]Merchant, error)
189.    UpdateMerchant(ctx context.Context, merch Merchant) (string, error)
190.    DeleteMerchant(ctx context.Context, id int) (string, error)
191.}
```

Inside the interface, every feature .needs to be listed and declared exactly how it will be coded. The parameter and return of every function have to be listed correctly as well.

The difference between monolith and microservice is in the database connection. In microservice, function which handles repository and query placed in one package with logic and everything else. But, the logic function cannot directly call the repository function. Like service interface, all repository function needs to be listed. This process applies to all services.

```
192.type Repository interface {
193.    InsertMerchantRepo(ctx context.Context, merch Merchant) error
194.    GetMerchantByIdRepo(ctx context.Context, id int) (Merchant, error)
195.    GetAllMerchantRepo(ctx context.Context) ([]Merchant, error)
196.    UpdateMerchantRepo(ctx context.Context, merch Merchant) error
197.    DeleteMerchantRepo(ctx context.Context, id int) error
198.}
```

After listing the repository, the real function could be called. Because both architectures serve the same purpose, the repository function is similar.

```
199.type repo struct {
200.    db     *sql.DB
201.    logger log.Logger
202.}
203.
204.func NewRepo(db *sql.DB, logger log.Logger) Repository {
205.    return &repo{
206.        db:     db,
207.        logger: log.With(logger, "repo", "sql"),
208.    }
209.}
210.
211.func  (repo  *repo)  InsertMerchantRepo(ctx  context.Context,  merch
   Merchant) error {
212.    sql := `INSERT INTO merchant (name)
213.    VALUES ($1)`
214.
215.    _, err := repo.db.ExecContext(ctx, sql, merch.Name)
216.    if err != nil {
217.        return err
218.    }
219.
220.    return nil
221.
222.}
```

As shown above, on line 211 the repository function uses context to communicate with another function. Line 199 until 209 purposes are to make repository function can receive pointer SQL DB and create log report. Outside the parameter and struct declared, the function query and logic are very similar.

Like use-case, the logic layer contains all the business logic functions. They are also responsible to call the repository function even though the method is different. Below is the code of InsertMerchant function

```
223.type service struct {
224.    repo   Repository
225.    logger log.Logger
226.}
227.
228.func NewService(rep Repository, logger log.Logger) Service {
229.    return &service{
230.        repo:   rep,
231.        logger: logger,
232.    }
233.}
234.func  (s  service)  InsertMerchant(ctx  context.Context,  merch  Merchant)
   (string, error) {
235.    var begin time.Time
236.    begin = time.Now()
237.
238.    err := s.repo.InsertMerchantRepo(ctx, merch)
239.     logger  := log.With(s.logger, "method", "Insert Merchant", "took",
   time.Since(begin))
240.    if err != nil {
241.        level.Error(logger).Log("err", err)
242.        return "Fail", err
243.    }
244.
245.      logger.Log("status",  "Successfully  insert  Merchant",  "took",
   time.Since(begin))
246.    return "Success", nil
247.}
```

On line 223, struct service is declared. This service struct is used to call the repo function since it contains a repository interface struct. The InsertMerchant logic function contains very much the same logical function with monolith, the difference is a logger in microservice handled by logic.

The endpoint layer is responsible to call the logic layer and pass them to the transport layer. All of the logic functions the service have to be listed on the endpoint. The endpoint is also responsible to check struct whether the request struct is already as expected or not. The code below shows one merchant service endpoint.

```
248.type Endpoints struct {
249.    InsertMerchant  endpoint.Endpoint
250.    GetMerchantById endpoint.Endpoint
251.    GetAllMerchant  endpoint.Endpoint
252.    UpdateMerchant  endpoint.Endpoint
253.    DeleteMerchant  endpoint.Endpoint
254.}
255.
256.func MakeEndpoint(s Service) Endpoints {
257.    return Endpoints{
258.        InsertMerchant:  makeInsertMerchantEndpoint(s),
259.        GetMerchantById: makeGetMerchantByIdEndpoint(s),
260.        GetAllMerchant:  makeGetAllMerchantEndpoint(s),
261.        UpdateMerchant:  makeUpdateMerchantEndpoint(s),
262.        DeleteMerchant:  makeDeleteMerchantEndpoint(s),
263.    }
264.}
265.func makeInsertMerchantEndpoint(s Service) endpoint.Endpoint {
266.    return func(ctx context.Context, request interface{}) (interface{},
   error) {
267.        var merch Merchant
268.        req := request.(InsertMerchantRequest)
269.        merch.Name = req.Name
270.        ok, err := s.InsertMerchant(ctx, merch)
271.        return InsertMerchantResponse{Ok: ok}, err
272.    }
273.}
```

From line number 248 until 254, all the features the service has listed on the endpoint struct. In Go-Kit, the endpoint has its own function where it calls the logic function and creates the response as shown from line 265 until 273.

The transport layer communicates with the outside world and to fulfill that purpose, the transport layer has a router and middleware. This router works to create API, encode response and decode the request. Below is the snippet code of the merchant service endpoint.

```
274.func    NewHTTPServer(ctx    context.Context,    endpoints    Endpoints)
   http.Handler {
275.    r := mux.NewRouter()
276.    r.Use(commonMiddleware)
277.
278.     r.Methods("POST").Path("/merchant/add").Handler(httptransport.NewS
   erver(
279.        endpoints.InsertMerchant,
280.        decodeMerchantReq,
281.        encodeResponse,
282.    ))
283.    return r
284.}
```

On line 275, a mux router was created. This router handles the HTTP request including responding to the request. On line 279 endpoint function is called. Therefore the logic could be executed. Line 280 purpose is to decode JSON from the request body and put the data to the expected struct. Line 281 is responsible to create the struct given to become JSON for the return.

On the main package, the DB connection function is called and passed to the function. Since this project builds 3 services, there are 3 DB connections. Every service also have their own port as shown on code below.

```
285.go func() {
        fmt.Println("User service listening on port", *httpAddr)
        handler := user.NewHTTPServer(ctx, endpoints)
        errs <- http.ListenAndServe(":8080", handler)
    }()
        go func() {
            fmt.Println("Merchant service listening on port :8081")
            handler2 := merchant.NewHTTPServer(ctx2, endpoints2)
            errs <- http.ListenAndServe(":8081", handler2)
        }()
    go func() {
        fmt.Println("Transaction service listening on port :8082")
        handler3 := transaction.NewHTTPServer(ctx3, endpoints3)
        errs <- http.ListenAndServe(":8082", handler3)
    }()
```

Every service runs independently on its port. This is done to make services don't rely upon one another. The process above applies to every service.

## 5.2. Results

To find out which architecture is better, several testing scenarios were conducted. The first testing scenario is to test all features from both architectures using the first testing design. In this design, every service, database, NGINX, and JMeter are inside a single device. In this step, JMeter hit both architecture's API.

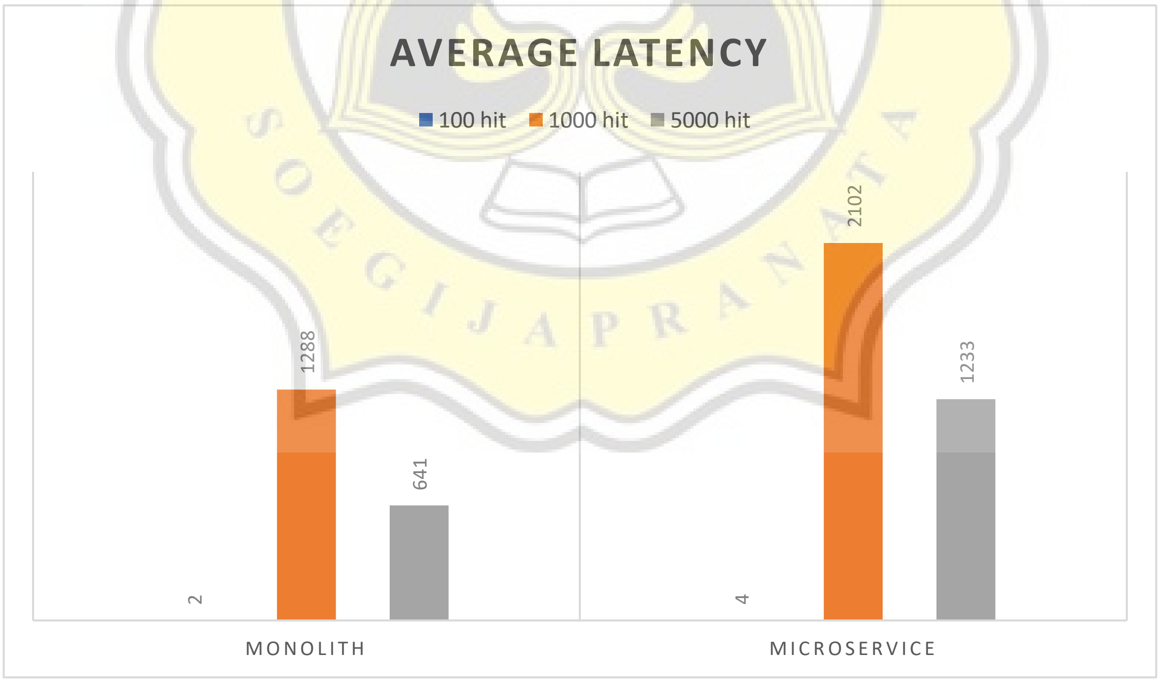


**Figure 5.1** Monolith Versus Microservice Merchant Get Service Performance

From the chart above it may seem that monolith architecture performs better. To make a more detailed comparison, statistical data from JMeter is shown in table 5.2 below

**Table 5.1.** Detailed Get Merchant Performance Data on Both Architecture

| **Features** | **Architecture** | | | | | |
|---|---|---|---|---|---|---|
| | **Monolith** | | | **Microservice** | | |
| **Total Request** | **100** | **1000** | **5000** | **100** | **1000** | **5000** |
| **Avg. Latency** | 2 ms | 1288 ms | 641 ms | 4 ms | 2102 ms | 1233 ms |
| **Success Rate** | 100% | 86.30% | 24.40% | 100% | 85.80% | 26.74% |
| **Max Latency** | 15 ms | 2909 ms | 5399 ms | 9 ms | 4792 ms | 10164 ms |

From the test evidence, while handling the smaller and medium amount of requests, the monolith application performs better while having average latency of 2ms. Microservice slightly edge the monolith only when handling 5000 requests on success rate. This poor performance by microservice could be caused by NGINX load time. This is proven from the terminal log where it shows that there is approximately a 2-millisecond delay between Jmeter latency and the terminal log latency. Meanwhile, the monolith application log shows that almost no delay between the terminal and Jmeter data. In the next step, this project test the insert feature. In this step, JMeter creates a POST HTTP request to the server in 1-second concurrently. Below is the request body looks.

```
300.POST http://127.0.0.1:10000/merchant/add
301.POST data:
302.{"name":"merchant1"}
```

For POST HTTP requests, data need to be sent as a request body and written in JSON format as shown on line 302. The suffix, on the data merchant name, is given by creating a counter variable on JMeter. Below is the result of the test.

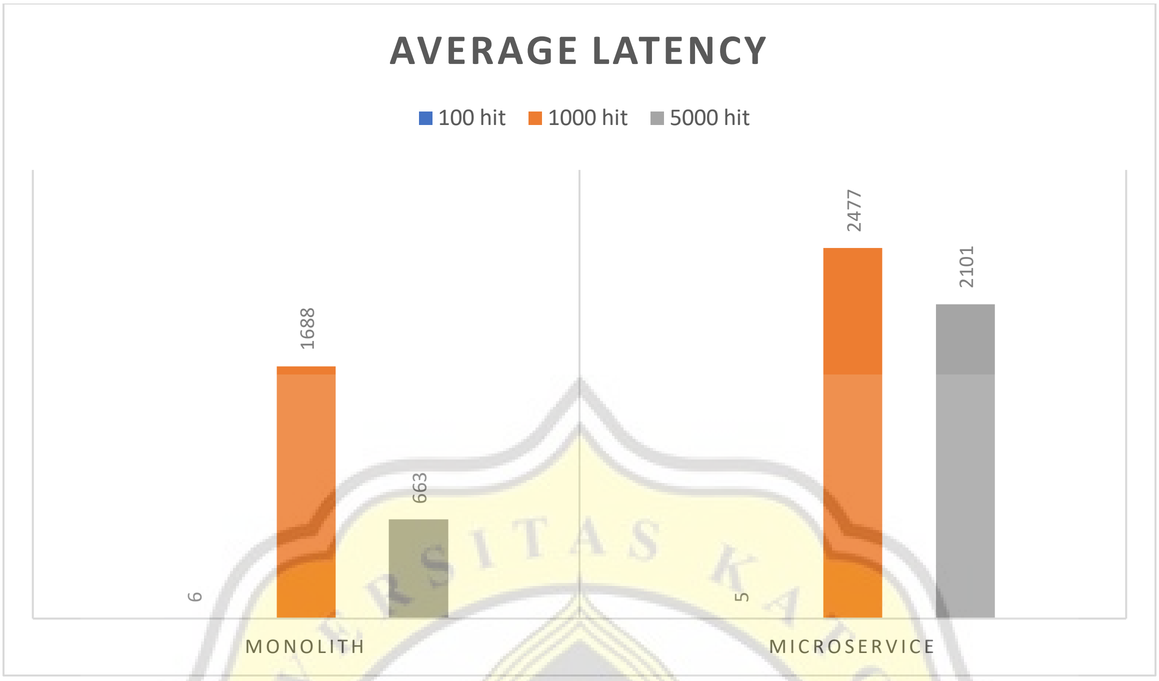


**Figure 5.2** Monolith Versus Microservice Insert Merchant Performance Chart

From the result above, there is an insane gap of latency between 100 hits and 1000 hits this could be caused by device limitation. This means that creating 1000 threads simultaneously takes many resources. To view more detailed results below is the data table

**Table 5.2.** Detailed Insert Merchant Performance Data on Both Architecture

| **Features** | **Architecture** | | | | | |
|---|---|---|---|---|---|---|
| | **Monolith** | | | **Microservice** | | |
| **Total Request** | **100** | **1000** | **5000** | **100** | **1000** | **5000** |
| **Avg. Latency** | 6 ms | 1688 ms | 663 ms | 5 ms | 2477 ms | 2101 ms |
| **Success Rate** | 100% | 65.20% | 18.02% | 100% | 99.50% | 25.84% |
| **Max Latency** | 116 ms | 2909 ms | 5399 ms | 101 ms | 4498 ms | 13004 ms |

From the data in table 5.2, it could be seen that surprisingly microservice edge monolith on 1000 hits. The success rate is far higher on every hit. The max latency and the average on monolith seem smaller but it caused by many failed requests. On the next step, the last merchant service feature, the update feature is tested to find out which architecture perform better when handling update request. Below is the request body data in JSON format sent.

```
{
    "idmerchant" : ${counter_value},
    "name" : "merchant${counter_value}"
}
```

The counter value will be replaced with a loop sequence number. Below is the result of the test.

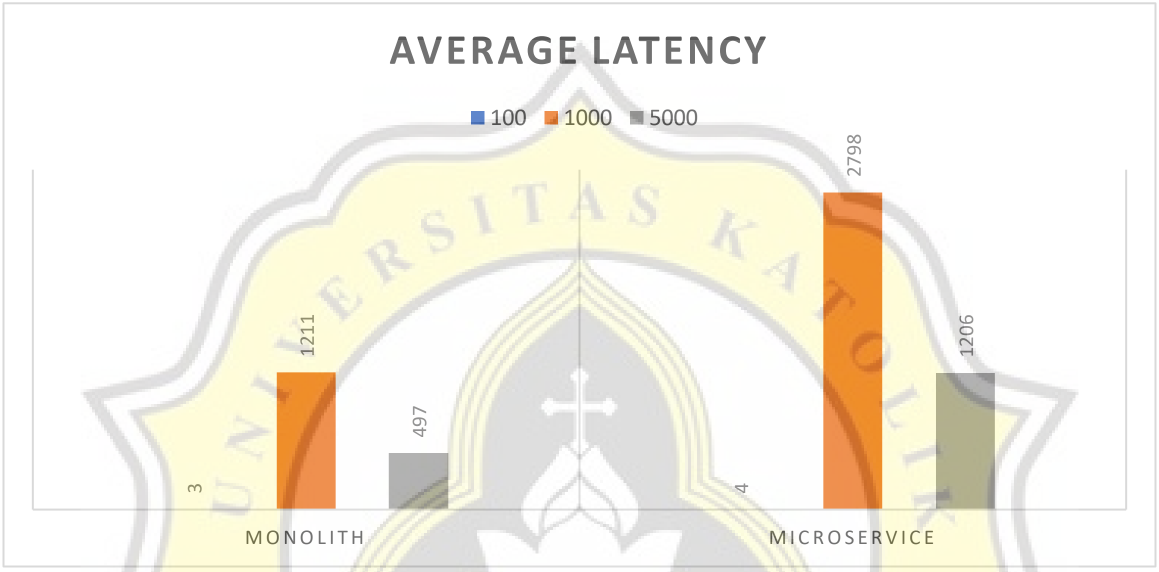


**Figure 5.3** Monolith Versus Microservice Update Merchant Performance Chart

From the chart above, monolith performance seems faster. But like any other test before, further observation needs to be done. Below is the detailed data of the test.

**Table 5.3.** Detailed Update Merchant Performance Data on Both Architecture

| **Features** | **Architecture** | | | | | |
|---|---|---|---|---|---|---|
| | **Monolith** | | | **Microservice** | | |
| **Total Request** | **100** | **1000** | **5000** | **100** | **1000** | **5000** |
| **Avg. Latency** | 3 ms | 1211 ms | 497 ms | 4 ms | 2798 ms | 1206 ms |
| **Success Rate** | 100% | 54.00% | 14.56% | 100% | 79.90% | 20.78% |
| **Max Latency** | 10 ms | 2909 ms | 8255 ms | 15 ms | 5584 ms | 10825 ms |

From table 5.3, it could be concluded that even though monoliths have better performance, the number of failed requests is also higher than microservice. On the other hand, in terms of handling the higher request, microservice have a higher success rate. The max latency indicates the highest latency thread accept for a single request. In terms of max latency, microservice has a bit worse record.

To test the application with higher complexity, in this step user service is tested to find out which architecture handles authentication better. In user service, the register feature has password encryption using the bcrypt algorithm. JMeter sends the request like below to the register's API.

```
{
      "email" : "user${counter}",
      "password" : "user${counter}"
}
```

This email will be added with a number sequence from the counter variable. Below is the result of the test.

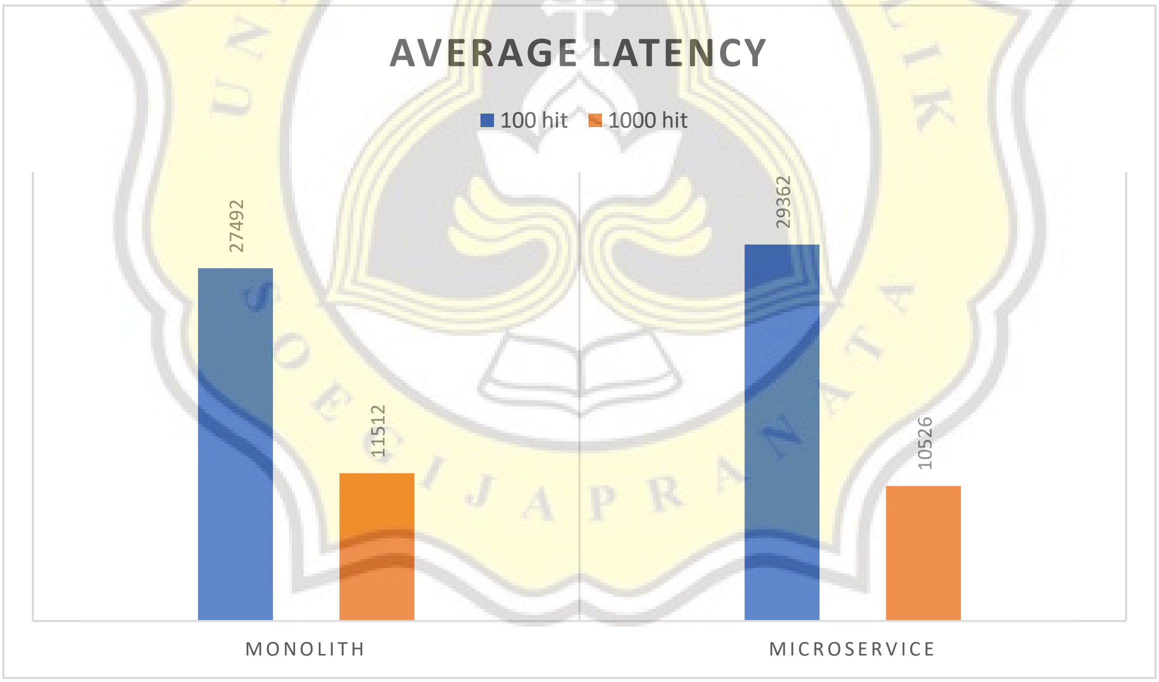


**Figure 5.4** Monolith Versus Microservice Register User Performance Chart

Both performance tests result very poorly with an average latency of over ten thousand milliseconds. From the chart, it seems that monolith performs better while handling 100 requests. But microservice handles larger performance better. The amount of requests sent is lower than the previous test because the number of errors while handling 1000 requests is already massive. To understand the test result better, it can see in the below table.

**Table 5.4.** Detailed Register User Performance Data on Both Architecture

| **Features** | **Architecture** | | | |
|---|---|---|---|---|
| | **Monolith** | | **Microservice** | |
| **Total Request** | **100** | **1000** | **100** | **1000** |
| **Avg. Latency** | 27492 ms | 11512 ms | 29362 ms | 10526 ms |
| **Success Rate** | 100% | 23.20% | 100% | 13.60% |
| **Max Latency** | 35241 ms | 83125 ms | 39607 ms | 60003 ms |

From the test result, unlike merchant testing results, microservice have a lower success rate during handling a larger amount of register requests.

In the next step, the login feature is tested. The login feature is the most complex because it checks the data from the database, finds the match data, compares the password, and creates JWT as a return. To find out which architecture handles login better can be seen on the chart below.

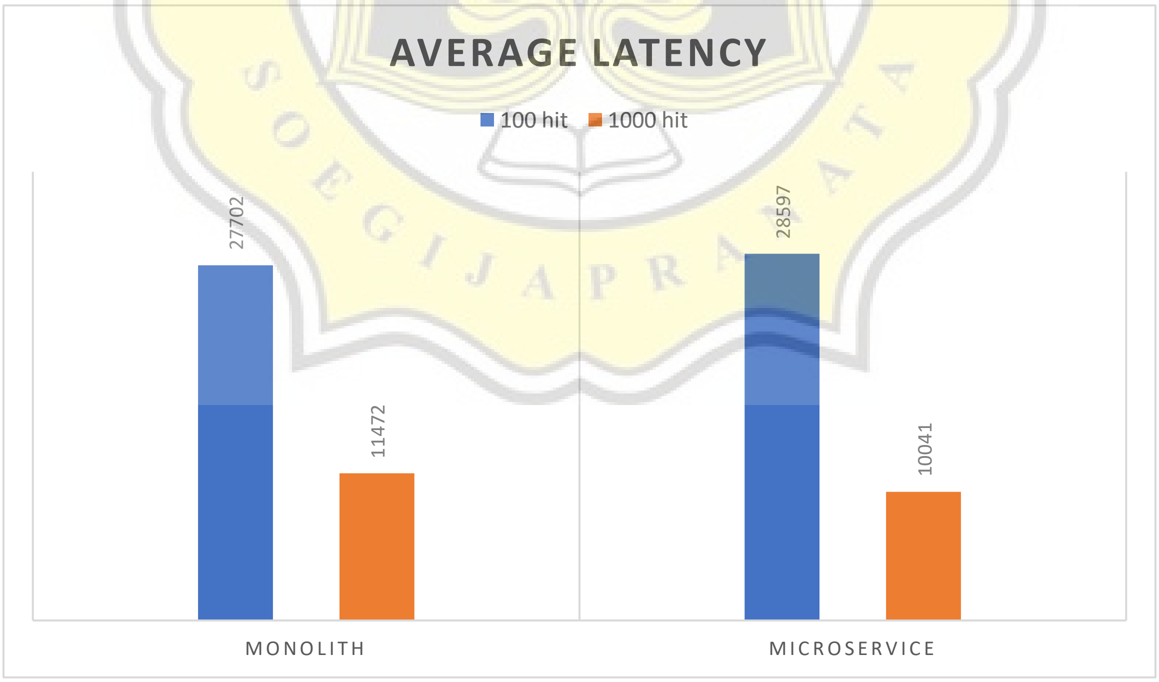


**Figure 5.5** Monolith Versus Microservice Login User Performance Chart

**Table 5.5.** Detailed Login User Performance Data on Both Architecture

| Features | Architecture | | | |
|---|---|---|---|---|
| | Monolith | | Microservice | |
| **Total Request** | **100** | **1000** | **100** | **1000** |
| **Avg. Latency** | 27702 ms | 11742 ms | 28597 ms | 10041 ms |
| **Success Rate** | 100% | 22.50% | 100% | 13.10% |
| **Max Latency** | 35241 ms | 83125 ms | 39607 ms | 60003 ms |

From the test result, both designs have a massive latency. This perhaps happened because the service is complex and the service needs time to complete. But, the result shows that monolith performs better with lower latency on 100 requests and a higher success rate on a larger amount of requests.

The last service that needs to be tested is the transaction service. Transaction service is a rather less complex computation than user service. This service contains bulk insert as the key feature. In this test, the struct used for the request can be seen below.

```
{
     "payment" : "cash",
    "detail": [
         {
         "productCode" : "Ikan",
         "quantity" : 10,
         "price" : 20000
         },
         {
             "productCode" : "Tahu",
             "quantity" : 10,
             "price" : 20000
         }]
}
```

This request has an array attribute that will be inserted on a different table. For saving resources used, the array will only contain two structs for each request. Below is the test result.

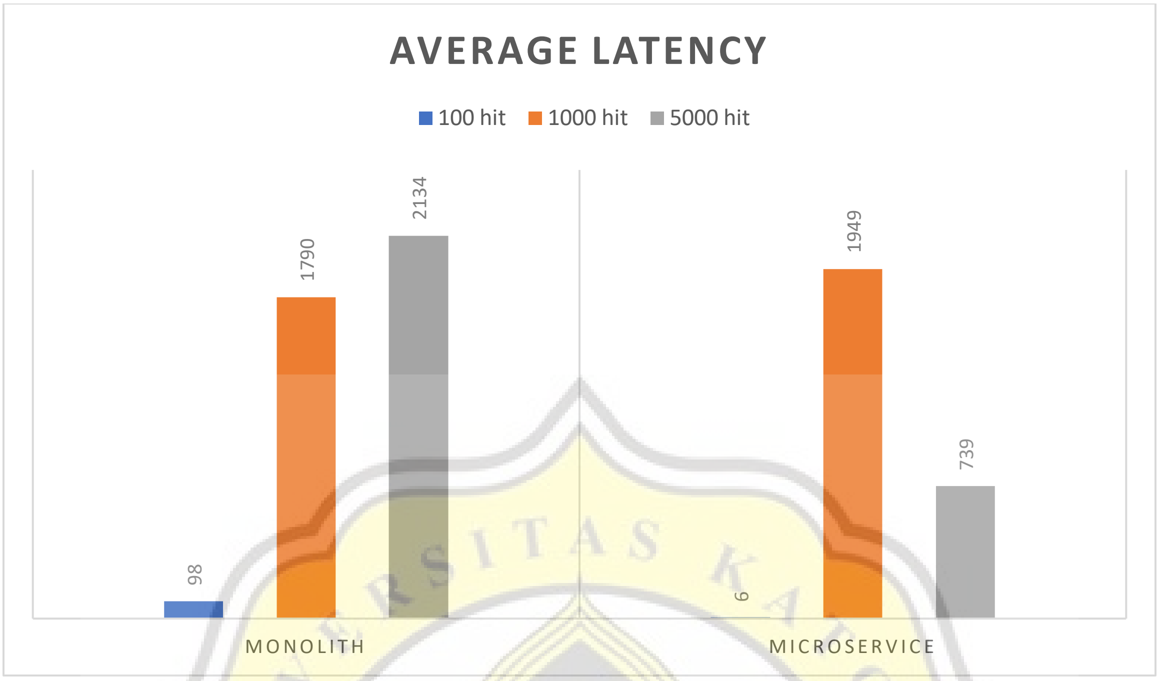


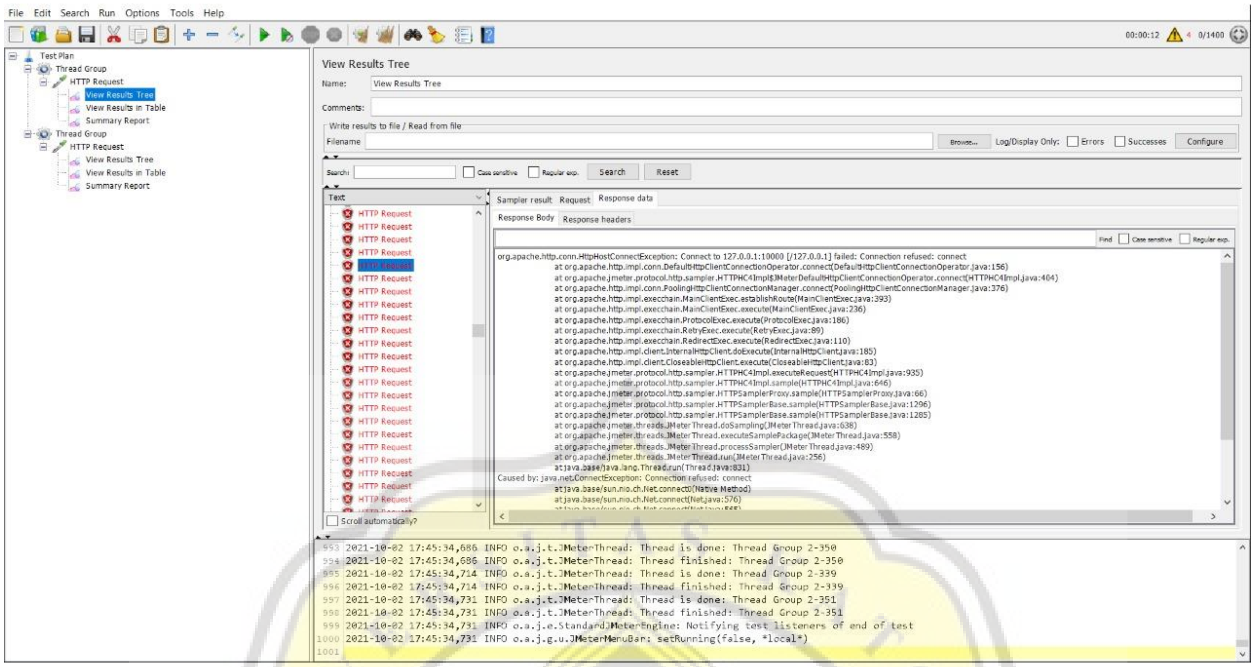
**Figure 5.6** Monolith Versus Microservice Insert Bulk Transaction Performance Chart

**Table 5.6.** Detailed Insert Bulk Transaction Performance Data on Both Architecture

| **Features** | **Architecture** | | | | | |
|---|---|---|---|---|---|---|
| | **Monolith** | | | **Microservice** | | |
| **Total Request** | **100** | **1000** | **5000** | **100** | **1000** | **5000** |
| **Avg. Latency** | 98 ms | 1790 ms | 2134 ms | 6 ms | 1849 ms | 739 ms |
| **Success Rate** | 100% | 50.40% | 30.82% | 100% | 70.10% | 20.00% |
| **Max Latency** | 402 ms | 6949 ms | 8255 ms | 33 ms | 5739 ms | 9803 ms |

From the test result, it could be concluded that in case handling insert transaction, microservice performs slightly better. But when it comes to handling a bigger amount of requests, the number of failures arise in microservice since it only achieves a 20 percent success rate. Although it seems that both architectures flop when handling a large number of requests, it can seem that in this particular service, microservice is slightly better.

The overall result of the test is worse than expected. There are too many errors on the test. The error message could be seen in figure 5.7 below



```
org.apache.http.conn.HttpHostConnectException: Connect to 127.0.0.1:10000 [/127.0.0.1] failed: Connection refused: connect
	at org.apache.http.impl.conn.DefaultHttpClientConnectionOperator.connect(DefaultHttpClientConnectionOperator.java:156)
	at org.apache.jmeter.protocol.http.sampler.HTTPHC4Impl$JMeterDefaultHttpClientConnectionOperator.connect(HTTPHC4Impl.java:404)
	at org.apache.http.impl.conn.PoolingHttpClientConnectionManager.connect(PoolingHttpClientConnectionManager.java:376)
	at org.apache.http.impl.execchain.MainClientExec.establishRoute(MainClientExec.java:393)
	at org.apache.http.impl.execchain.MainClientExec.execute(MainClientExec.java:236)
	at org.apache.http.impl.execchain.ProtocolExec.execute(ProtocolExec.java:186)
	at org.apache.http.impl.execchain.RetryExec.execute(RetryExec.java:89)
	at org.apache.http.impl.execchain.RedirectExec.execute(RedirectExec.java:110)
	at org.apache.http.impl.client.InternalHttpClient.doExecute(InternalHttpClient.java:185)
	at org.apache.http.impl.client.CloseableHttpClient.execute(CloseableHttpClient.java:83)
	at org.apache.jmeter.protocol.http.sampler.HTTPHC4Impl.executeRequest(HTTPHC4Impl.java:935)
	at org.apache.jmeter.protocol.http.sampler.HTTPHC4Impl.sample(HTTPHC4Impl.java:646)
	at org.apache.jmeter.protocol.http.sampler.HTTPSamplerProxy.sample(HTTPSamplerProxy.java:66)
	at org.apache.jmeter.protocol.http.sampler.HTTPSamplerBase.sample(HTTPSamplerBase.java:1296)
	at org.apache.jmeter.protocol.http.sampler.HTTPSamplerBase.sample(HTTPSamplerBase.java:1285)
	at org.apache.jmeter.threads.JMeterThread.doSampling(JMeterThread.java:638)
	at org.apache.jmeter.threads.JMeterThread.executeSamplePackage(JMeterThread.java:558)
	at org.apache.jmeter.threads.JMeterThread.processSampler(JMeterThread.java:489)
	at org.apache.jmeter.threads.JMeterThread.run(JMeterThread.java:256)
	at java.base/java.lang.Thread.run(Thread.java:831)
Caused by: java.net.ConnectException: Connection refused: connect
	at java.base/sun.nio.ch.Net.connect0(Native Method)
	at java.base/sun.nio.ch.Net.connect(Net.java:576)
```

```
2021-10-02 17:45:34,686 INFO o.a.j.t.JMeterThread: Thread is done: Thread Group 2-350
2021-10-02 17:45:34,686 INFO o.a.j.t.JMeterThread: Thread finished: Thread Group 2-350
2021-10-02 17:45:34,714 INFO o.a.j.t.JMeterThread: Thread is done: Thread Group 2-339
2021-10-02 17:45:34,714 INFO o.a.j.t.JMeterThread: Thread finished: Thread Group 2-339
2021-10-02 17:45:34,731 INFO o.a.j.t.JMeterThread: Thread is done: Thread Group 2-351
2021-10-02 17:45:34,731 INFO o.a.j.t.JMeterThread: Thread finished: Thread Group 2-351
2021-10-02 17:45:34,731 INFO o.a.j.e.StandardJMeterEngine: Notifying test listeners of end of test
2021-10-02 17:45:34,731 INFO o.a.j.g.u.JMeterMenuBar: setRunning(false, *local*)
```

**Figure 5.7** Error Message From JMeter Failed Thread

When the message appears, it turns out that the server machine is still running. Also from monitoring the terminal log, it appears that the server did not face any error. Meanwhile, NGINX facing the same error as JMeter shows that the server is refusing the request.

```
325.2021/10/04 19:43:19 [error] 17544#15600: *226 connect() failed (10061:
    No connection could be made because the target machine actively refused
    it) while connecting to upstream, client: 127.0.0.1, server: localhost,
    request: "GET /api/merchant/list HTTP/1.1", upstream:
    "http://[::1]:8081/merchant/list", host: "127.0.0.1:9090"
326.2021/10/04 19:43:19 [error] 17544#15600: *226 no live upstreams while
    connecting to upstream, client: 127.0.0.1, server: localhost, request:
    "GET /api/merchant/list HTTP/1.1", upstream:
    "http://localhost/merchant/list", host: "127.0.0.1:9090"
```

From the observation above, moving JMeter to another device perhaps could be the solution. In the second testing design, JMeter is separated onto another device since the JMeter user interface and service already used a huge amount of RAM usage. For the second design, only the key features were tested. This is done to find out whether device limitations have an aspect of the failure rate. But moving the JMeter to another device means that the request has more steps from client to server.

The first comparison is between the Get Merchant feature. All the method are the same but, in this test, JMeter sends a request from another device and hit the IP address of the server. Below is the result of the test.

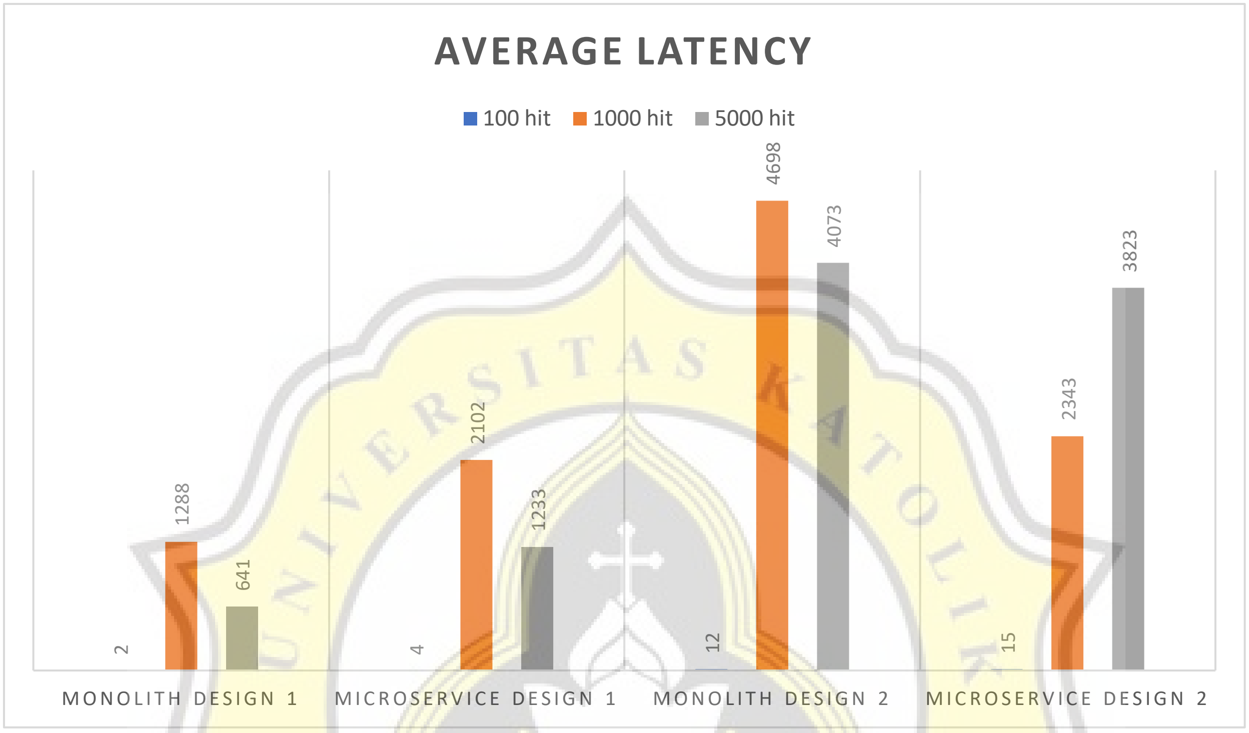


**Figure 5.8** Design Test 1 VS Design Test 2 Get Merchant Performance Chart

**Table 5.7.** Detailed Get Merchant Performance Data on Both Architecture

| **Features** | **Architecture** | | | | | |
|---|---|---|---|---|---|---|
| | **Monolith** | | | **Microservice** | | |
| **Total Request** | **100** | **1000** | **5000** | **100** | **1000** | **5000** |
| **Avg. Latency** | 12 ms | 4698 ms | 4073 ms | 15 ms | 2343 ms | 3823 ms |
| **Success Rate** | 100% | 72.70% | 35.70% | 100% | 52.80% | 10.04% |
| **Max Latency** | 31 ms | 6650ms | 11115 ms | 106 ms | 5965 ms | 10970 ms |

From the test result above, it seems that the performance becomes poorer in the second design. In success rate wise, for this get merchant feature alose worse. But it won't be fair to only test one feature. The heaviest service needs to be tested as well. In this step, the user login service is tested using a second design. Below is the result of the test.

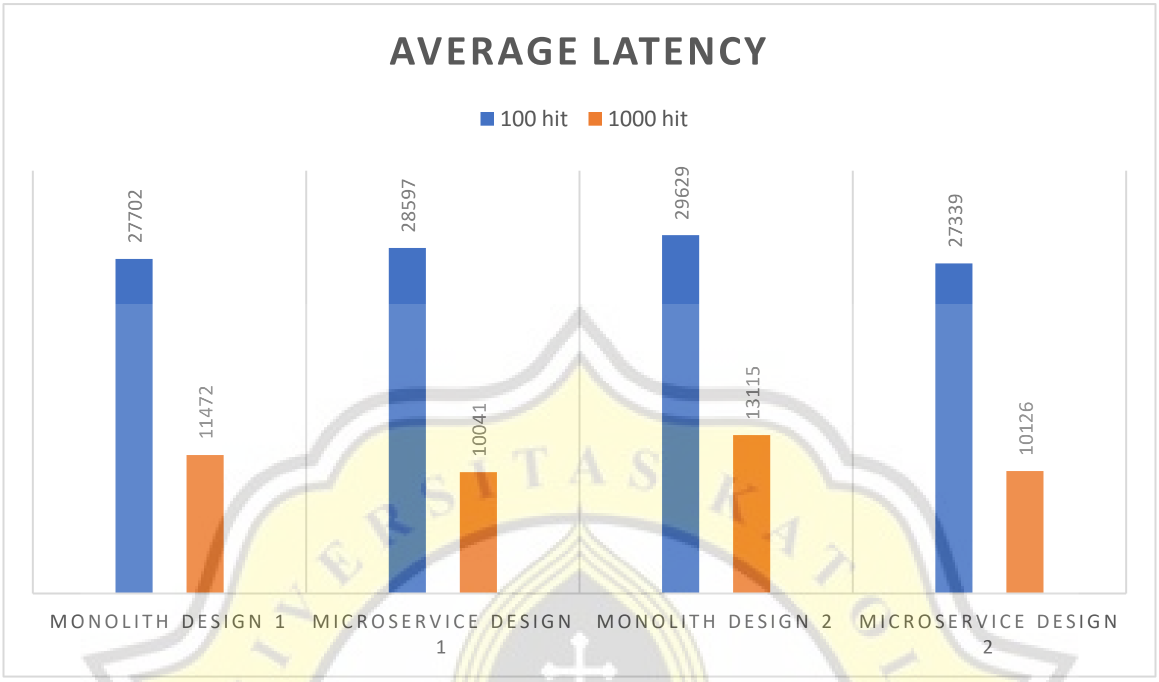


**Figure 5.9** Design 1 Versus Design 2 Login User Performance Chart

**Table 5.8.** Detailed Login User Performance Data on Both Architecture

| **Features** | **Architecture** | | | |
|---|---|---|---|---|
| | **Monolith** | | **Microservice** | |
| **Total Request** | **100** | **1000** | **100** | **1000** |
| **Avg. Latency** | 29629 ms | 13115 ms | 273329 ms | 10126 ms |
| **Success Rate** | 100% | 22.30% | 100% | 13.50% |
| **Max Latency** | 37844 ms | 88281 ms | 42521 ms | 60151 ms |

From figure 5.9 it could indicate that design 2 latency is slightly worse. Not only that, this test indicates that the error rate from design 1 is slightly lower for both architecture than design 2. The insert transaction feature is tested to find out the result. Below is the result of the test.

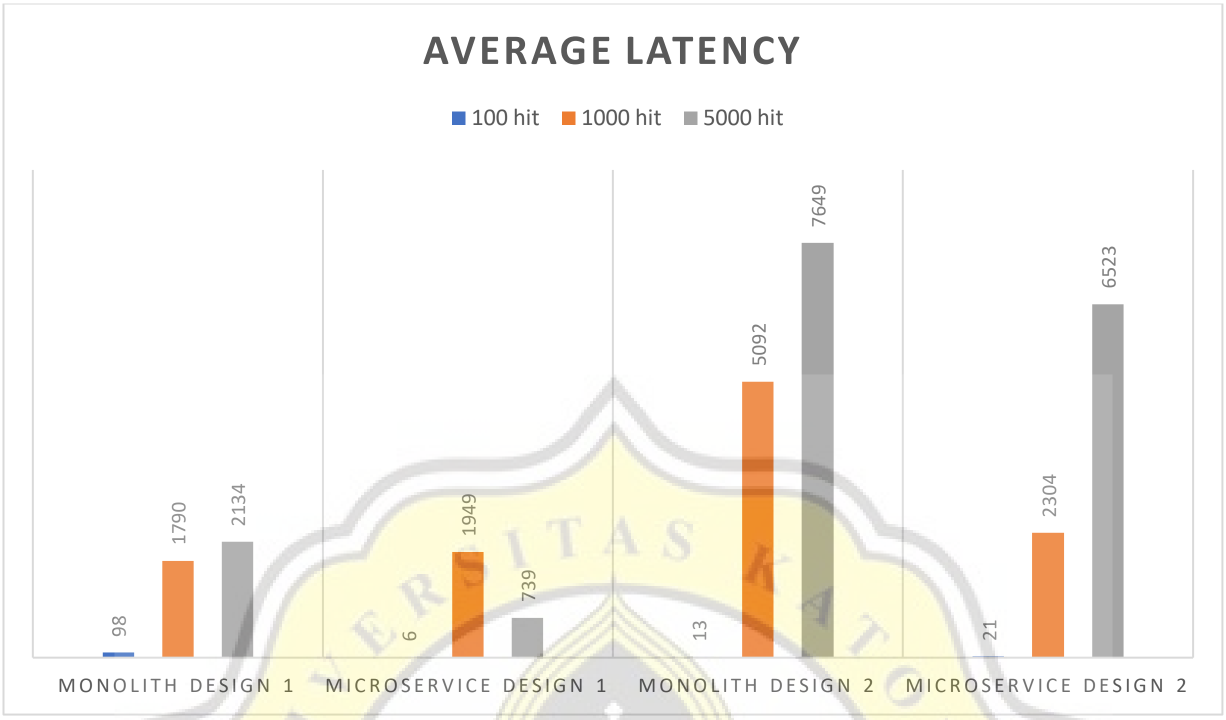


**Figure 5.10** Monolith Versus Microservice Insert Bulk Transaction Performance Chart

**Table 5.9.** Detailed Insert Bulk Transaction Performance Data on Both Architecture

| **Features** | **Architecture** | | | | | |
|---|---|---|---|---|---|---|
| | **Monolith** | | | **Microservice** | | |
| **Total Request** | **100** | **1000** | **5000** | **100** | **1000** | **5000** |
| **Avg. Latency** | 13 ms | 5092 ms | 7649 ms | 21 ms | 2304 ms | 6523 ms |
| **Success Rate** | 100% | 52.30% | 23.52% | 100% | 34.50% | 20.14% |
| **Max Latency** | 41 ms | 11870 ms | 22260 ms | 123 ms | 6859 ms | 29908 ms |

From the chart, in figure 5.10 it could be seen that the performance gap between the first design and the second design is huge. With this kind of gap, it is safe to say that the performance from the second design is worse.

Because it seems that the microservice doesn't live up to the expectation, to minimize ram usage and in an attempt to improve the performance of microservice, NGINX is separated to another device with an IP address located at 192.168.1.22. In this step, only microservice features are tested since, in this third design, JMeter still hits monolith architecture directly.

Just like all the tests done before, Jmeter will hit the API provided by the server. But this time, unlike the second design, JMeter will hit 192.168.1.22. The first test conducted is to see how is the performance of the third design while handling merchant gets requests.

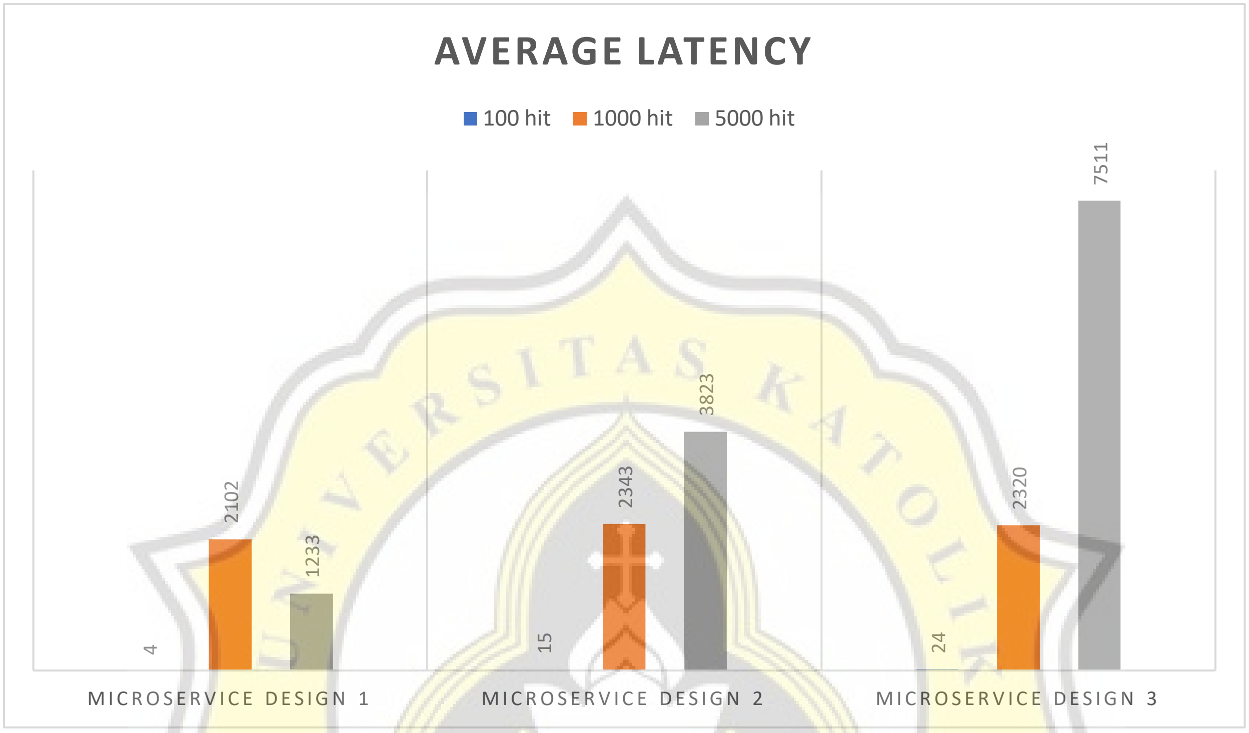


**Figure 5.11** Design Test 1 VS Design Test 2 and Design Test 3 Get Merchant Performance Chart

**Table 5.10.** Detailed Get Merchant Performance Data on Design Test 2 and Design Test 3

| **Features** | **Architecture** | | | | | |
|---|---|---|---|---|---|---|
| | **Design 2** | | | **Design 3** | | |
| **Total Request** | **100** | **1000** | **5000** | **100** | **1000** | **5000** |
| **Avg. Latency** | 15 ms | 2343 ms | 3823 ms | 24 ms | 2320 ms | 7511 ms |
| **Success Rate** | 100% | 52.80% | 10.04% | 100% | 99.30% | 58.26% |
| **Max Latency** | 106 ms | 5965 ms | 10970 ms | 135 ms | 4861 ms | 17187 ms |

Compared to the second design, the third design handle more requests poorly. But in terms of success rate, somehow third design is edging the second design by a huge gap. To add more weight to the comparison, the user login feature is tested, and below is the result from the test.

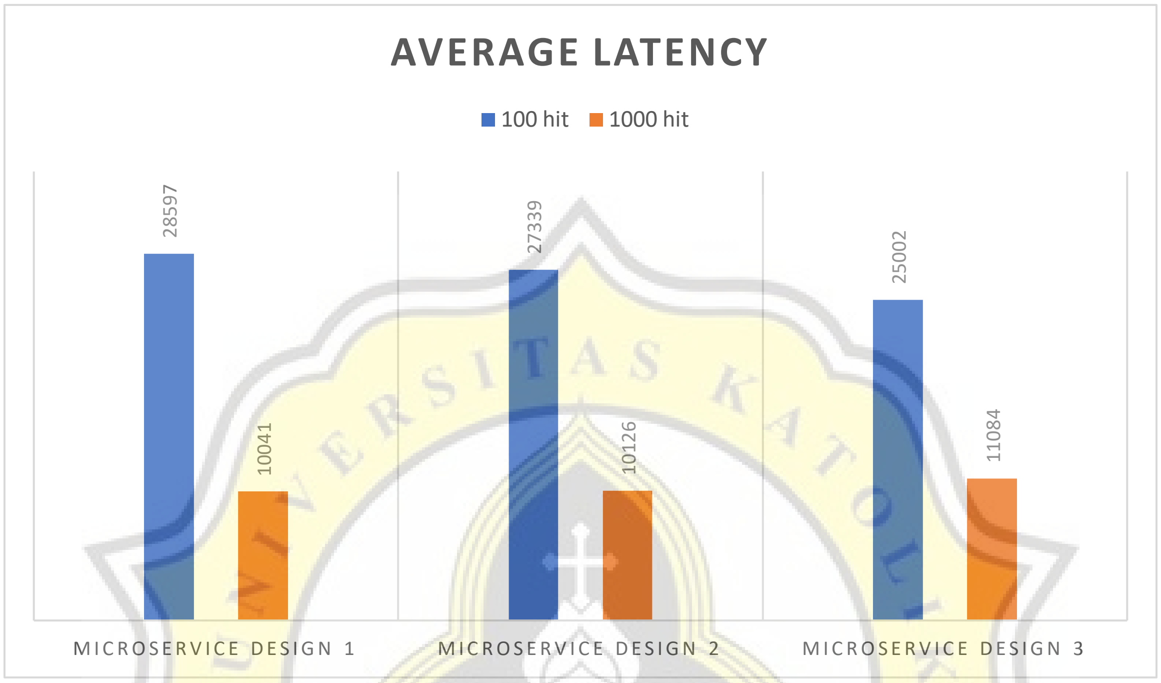


**Figure 5.12** Design 1 Versus Design 2 and Design 3 Login User Performance Chart

**Table 5.11.** Detailed Design 2 and Design 3 Login User Performance

| **Features** | **Architecture** | | | |
|---|---|---|---|---|
| | **Design 2** | | **Design 3** | |
| **Total Request** | **100** | **1000** | **100** | **1000** |
| **Avg. Latency** | 273329 ms | 10126 ms | 25002 ms | 11084 ms |
| **Success Rate** | 100% | 13.50% | 100% | 13.20% |
| **Max Latency** | 42521 ms | 60151 ms | 39539 ms | 60209 ms |

From the test result above, it seems that on small requests, the third design has the best performance but it is the opposite with a larger amount of requests where the third design only gets a 13.20% success rate. The final test is to compare the performance while handling the insert transaction. Below is the result of the test.

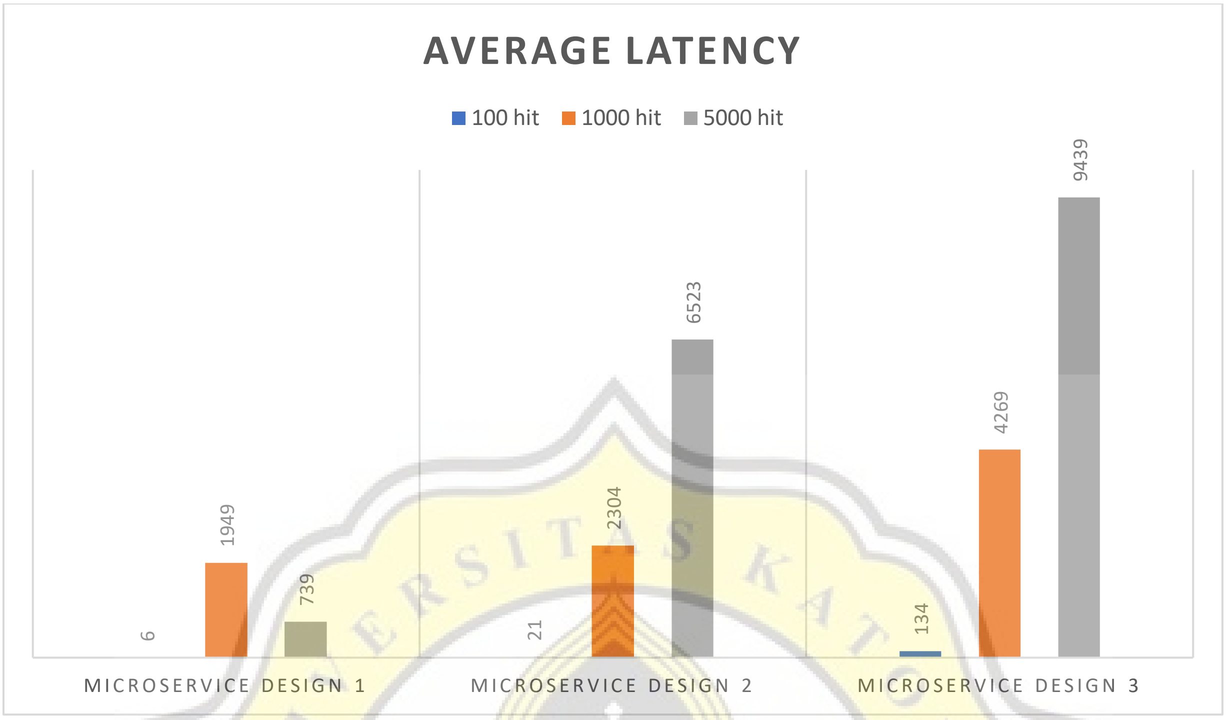


**Figure 5.13** Design 1 Versus Design 2 and Design 3 Insert Bulk Transaction Performance Chart

**Table 5.12.** Detailed Design 2 and Design 3 Insert Bulk Transaction Performance

| **Features** | **Architecture** | | | | | |
|---|---|---|---|---|---|---|
| | **Design 2** | | | **Design 3** | | |
| **Total Request** | **100** | **1000** | **5000** | **100** | **1000** | **5000** |
| **Avg. Latency** | 21 ms | 2304 ms | 6523 ms | 134 ms | 4269 ms | 9439 ms |
| **Success Rate** | 100% | 34.50% | 20.14% | 100% | 57.70% | 28.28% |
| **Max Latency** | 123 ms | 6859 ms | 29908 ms | 123 ms | 6859 ms | 29908 ms |

From the test result above, the performance of the third design has become slower as the request number increases. But in terms of success rate, the third design comes out victorious. When more requests are successfully sent, the average latency will be higher. This concludes that separating NGINX from another device is not boosting the performance but the success rate. The performance is worse because the request sent by the client has to go through the router, the NGINX device, and finally to the server.

**Table 5.13.** Overall Average Services Latency

| Services | Average Latency | | Success Rate | |
|---|---|---|---|---|
| | Monolith | Microservice | Monolith | Microservice |
| **Merchant** | 663,22 ms | 1325,56 ms | 62,50% | 70,95% |
| **User** | 19612,00 ms | 19640,50 ms | 61,43% | 56,68% |
| **Transaction** | 1340,67 ms | 864,67 ms | 60,41% | 63,37% |
| **TOTAL AVERAGE** | 7205,30 ms | 7276,91 ms | 61,44% | 63,66% |

To summarize all of the tests conducted, the average latency and success rate of every service summed up to find out the overall average as seen in table 5.13.

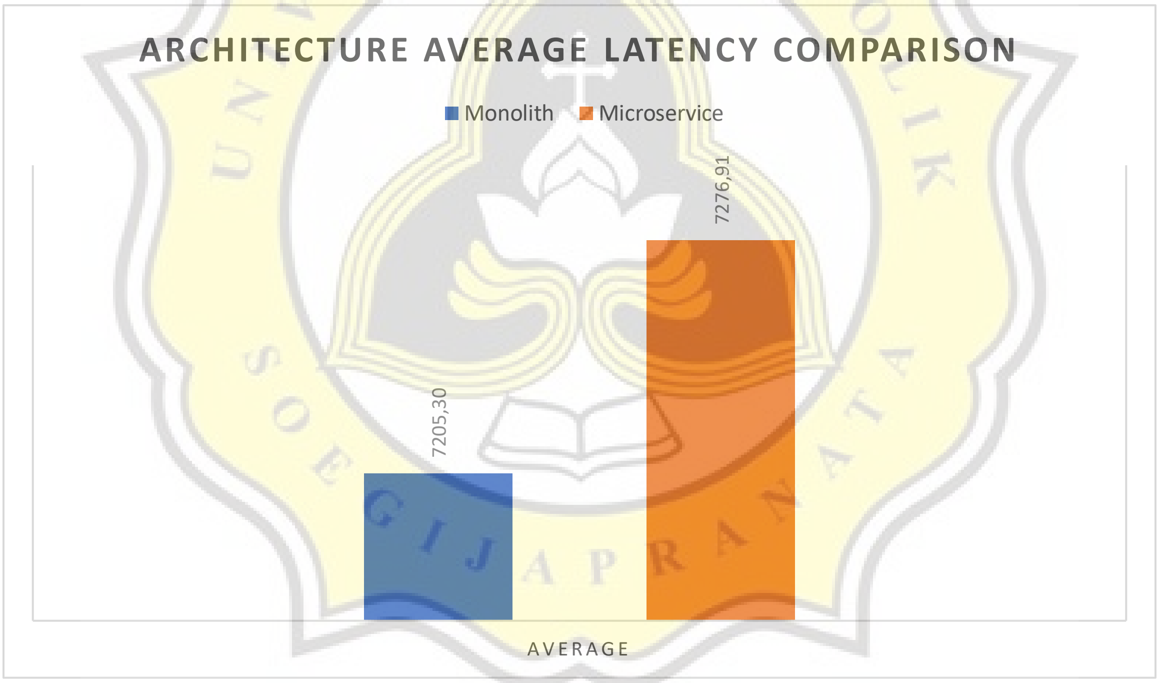


**Figure 5.14** Overall Monolith Versus Microservice Average Latency Chart

As seen in figure 5.14, in terms of average latency for every service, the monolith has better latency with a 71.61 ms gap difference.

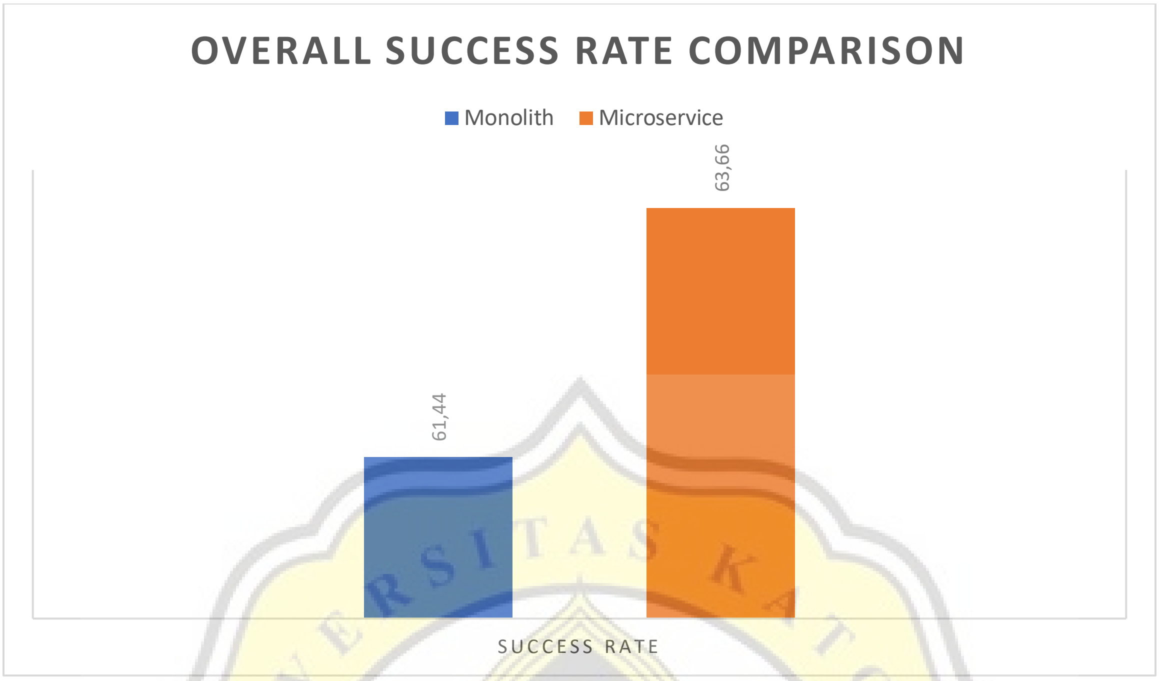


**Figure 5.15** Overall Monolith Versus Microservice Average Success Rate Chart

From figure 5.15, it could be seen that in terms of success rate, microservice slightly edge monolith average success rate with 2.22%.

**Table 5.14.** Overall Average Latency and Success Rate Every Test Design

| Services | Average Latency | | | Success Rate | | |
|---|---|---|---|---|---|---|
| | **Design 1** | **Design 2** | **Design 3** | **Design 1** | **Design 2** | **Design 3** |
| **Merchant** | 1113,00 | 2060,33 | 3285,00 | 70,85 | 54,28 | 85,85 |
| **User** | 19319,00 | 18732,50 | 18043,00 | 56,80 | 56,75 | 56,60 |
| **Transaction** | 898,00 | 2949,33 | 4614,00 | 63,37 | 51,55 | 61,99 |
| **Total Average** | 7110,00 | 7914,06 | 8647,33 | 63,67 | 54,19 | 68,15 |

From table 5.14, it could be seen that testing design is affecting performance for each service.

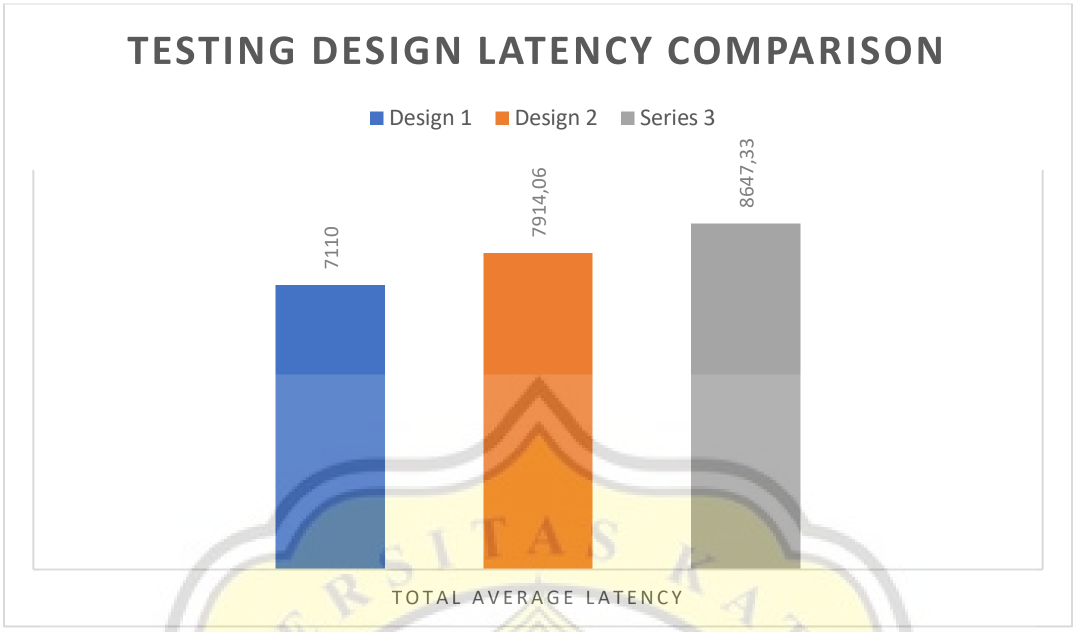


**Figure 5.16** Testing Design Latency Comparison Chart

From figure 5.16 it could be concluded that testing design affects average latency where from the chart design 1 has the best average latency and it becomes worse on design 2 and design 3.

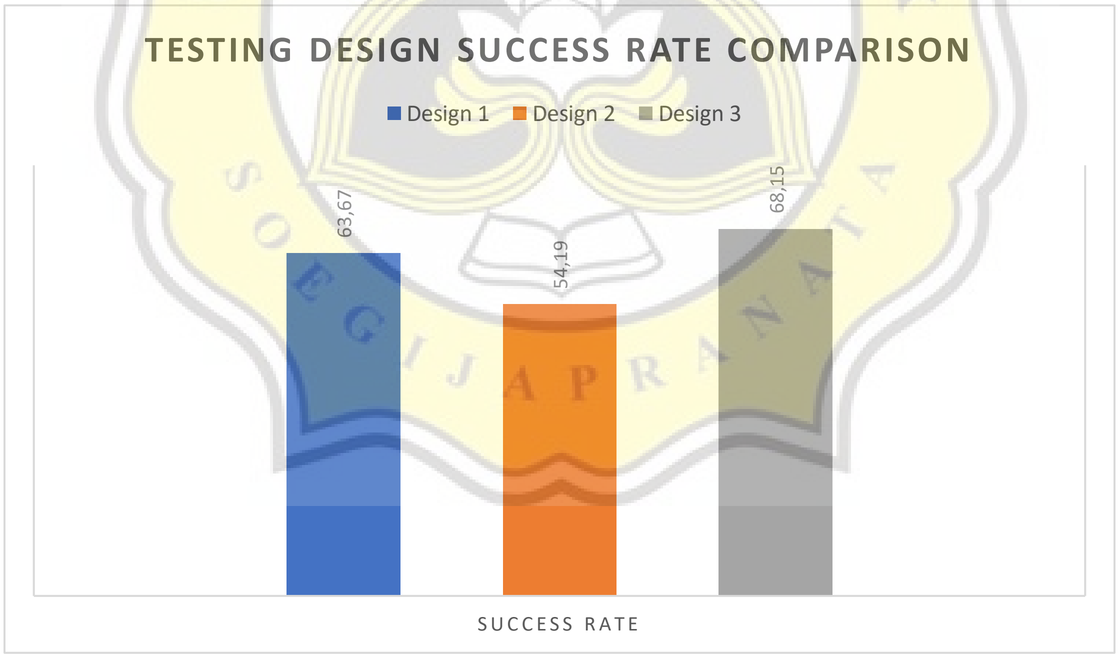


**Figure 5.17** Testing Design Success Rate Comparison Chart

As seen in figure 5.17, the success rate is affected by the testing design. Unlike average latency, the success rate is slightly better on design 3 where NGINX is separated.