

Withdrawn Draft

Warning Notice

The attached draft document has been withdrawn, and is provided solely for historical purposes. It has been superseded by the document identified below.

Withdrawal Date August 07, 2019

Original Release Date March 25, 2019

Superseding Document

Status Final

Series/Number NIST Special Publication 800-204

Title Security Strategies for Microservices-based Application Systems

Publication Date August 2019

DOI <https://doi.org/10.6028/NIST.SP.800-204>

CSRC URL <https://csrc.nist.gov/publications/detail/sp/800-204/final>

Additional Information N/A

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28

Security Strategies for Microservices-based Application Systems

Ramaswamy Chandramouli

This publication is available free of charge from:
<https://doi.org/10.6028/NIST.SP.800-204-draft>

C O M P U T E R S E C U R I T Y

29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64

Draft NIST Special Publication 800-204

Security Strategies for Microservices-based Application Systems

Ramaswamy Chandramouli
*Computer Security Division
Information Technology Laboratory*

This publication is available free of charge from:
<https://doi.org/10.6028/NIST.SP.800-204-draft>

March 2019



U.S. Department of Commerce
Wilbur L. Ross, Jr., Secretary

National Institute of Standards and Technology
Walter Copan, NIST Director and Under Secretary of Commerce for Standards and Technology

65
66
67
68
69
70
71
72
73
74

75
76
77
78
79
80
81
82
83
84
85
86
87
88

89
90
91
92

93
94
95
96
97
98

99
100
101

102
103
104
105
106
107
108
109
110
111

Authority

This publication has been developed by NIST in accordance with its statutory responsibilities under the Federal Information Security Modernization Act (FISMA) of 2014, 44 U.S.C. § 3551 *et seq.*, Public Law (P.L.) 113-283. NIST is responsible for developing information security standards and guidelines, including minimum requirements for federal information systems, but such standards and guidelines shall not apply to national security systems without the express approval of appropriate federal officials exercising policy authority over such systems. This guideline is consistent with the requirements of the Office of Management and Budget (OMB) Circular A-130.

Nothing in this publication should be taken to contradict the standards and guidelines made mandatory and binding on federal agencies by the Secretary of Commerce under statutory authority. Nor should these guidelines be interpreted as altering or superseding the existing authorities of the Secretary of Commerce, Director of the OMB, or any other federal official. This publication may be used by nongovernmental organizations on a voluntary basis and is not subject to copyright in the United States. Attribution would, however, be appreciated by NIST.

National Institute of Standards and Technology Special Publication 800-204
Natl. Inst. Stand. Technol. Spec. Publ. 800-204, 41 Pages (March 2019)
CODEN: NSPUE2

This publication is available free of charge from:
<https://doi.org/10.6028/NIST.SP.800-204-draft>

Certain commercial entities, equipment, or materials may be identified in this document in order to describe an experimental procedure or concept adequately. Such identification is not intended to imply recommendation or endorsement by NIST, nor is it intended to imply that the entities, materials, or equipment are necessarily the best available for the purpose.

There may be references in this publication to other publications currently under development by NIST in accordance with its assigned statutory responsibilities. The information in this publication, including concepts and methodologies, may be used by federal agencies even before the completion of such companion publications. Thus, until each publication is completed, current requirements, guidelines, and procedures, where they exist, remain operative. For planning and transition purposes, federal agencies may wish to closely follow the development of these new publications by NIST.

Organizations are encouraged to review all draft publications during public comment periods and provide feedback to NIST. Many NIST cybersecurity publications, other than the ones noted above, are available at <https://csrc.nist.gov/publications>.

Public comment period: *March 25, 2019 through April 26, 2019*

National Institute of Standards and Technology
Attn: Computer Security Division, Information Technology Laboratory
100 Bureau Drive (Mail Stop 8930) Gaithersburg, MD 20899-8930
Email: sp800-204-comments@nist.gov

All comments are subject to release under the Freedom of Information Act (FOIA).

112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155

Reports on Computer Systems Technology

The Information Technology Laboratory (ITL) at the National Institute of Standards and Technology (NIST) promotes the U.S. economy and public welfare by providing technical leadership for the Nation’s measurement and standards infrastructure. ITL develops tests, test methods, reference data, proof of concept implementations, and technical analyses to advance the development and productive use of information technology. ITL’s responsibilities include the development of management, administrative, technical, and physical standards and guidelines for the cost-effective security and privacy of other than national security-related information in Federal information systems. The Special Publication 800-series reports on ITL’s research, guidelines, and outreach efforts in information system security, and its collaborative activities with industry, government, and academic organizations.

Abstract

Microservices architecture is increasingly being used to develop application systems since its smaller codebase facilitates faster code development, testing, and deployment as well as optimization of the platform based on the type of microservice, support for independent development teams, and the ability to scale each component independently. Microservices generally communicate with each other using APIs, which requires several core features to support complex interactions between a substantial number of components. These core features include authentication and access management, service discovery, secure communication protocols, security monitoring, availability/resiliency improvement techniques (e.g., circuit breakers), load balancing and throttling, integrity assurance techniques during induction of new services, and handling of session persistence. Additionally, the core features could be bundled or packaged into architectural frameworks such as API gateways and service mesh. The purpose of this document is to analyze the multiple implementation options available for each individual core feature and configuration options in architectural frameworks, develop security strategies that counter threats specific to microservices, and enhance the overall security profile of the microservices-based application.

Keywords

microservices; load balancing; circuit breaker; Application Programming Interface (API); API gateway; service mesh; proxy.

Acknowledgements

<TBD>

Call for Patent Claims

156
157
158 This public review includes a call for information on essential patent claims (claims whose use
159 would be required for compliance with the guidance or requirements in this Information
160 Technology Laboratory (ITL) draft publication). Such guidance and/or requirements may be
161 directly stated in this ITL Publication or by reference to another publication. This call also
162 includes disclosure, where known, of the existence of pending U.S. or foreign patent applications
163 relating to this ITL draft publication and of any relevant unexpired U.S. or foreign patents.

164 ITL may require from the patent holder, or a party authorized to make assurances on its behalf, in
165 written or electronic form, either:

- 166 a) assurance in the form of a general disclaimer to the effect that such party does not hold
167 and does not currently intend holding any essential patent claim(s); or
- 168 b) assurance that a license to such essential patent claim(s) will be made available to
169 applicants desiring to utilize the license for the purpose of complying with the guidance
170 or requirements in this ITL draft publication either:
- 171 i. under reasonable terms and conditions that are demonstrably free of any unfair
172 discrimination; or
 - 173 ii. without compensation and under reasonable terms and conditions that are
174 demonstrably free of any unfair discrimination.

175 Such assurance shall indicate that the patent holder (or third party authorized to make assurances
176 on its behalf) will include in any documents transferring ownership of patents subject to the
177 assurance, provisions sufficient to ensure that the commitments in the assurance are binding on
178 the transferee, and that the transferee will similarly include appropriate provisions in the event of
179 future transfers with the goal of binding each successor-in-interest.

180 The assurance shall also indicate that it is intended to be binding on successors-in-interest
181 regardless of whether such provisions are included in the relevant transfer documents.

182 Such statements should be addressed to: sp800-204-comments@nist.gov.

Table of Contents

183
184
185 EXECUTIVE SUMMARYvi
186 1. INTRODUCTION, SCOPE, AND TARGET AUDIENCE1
187 1.1 Scope of this document1
188 1.2 Target Audience.....1
189 1.3 Relationship to other NIST Guidance Documents1
190 1.4 Methodology and Organization.....2
191 2. MICROSERVICES-BASED APPLICATION SYSTEMS – TECHNOLOGY BACKGROUND3
192 2.1 Microservices – a Conceptual View3
193 2.2 Microservices – Design Principles3
194 2.3 Business drivers.....4
195 2.4 Building Blocks.....4
196 2.5 Microservices – Interaction Styles.....5
197 2.6 Microservices – State of the Practice Core Features.....7
198 2.7 Microservices – Architectural Frameworks8
199 2.7.1 *API Gateway*9
200 2.7.2 *Service Mesh*10
201 2.8 Comparison with Monolithic Architecture11
202 2.9 Comparison with Service-Oriented Architecture (SOA).....11
203 2.10 Advantages of Microservices.....12
204 2.11 Disadvantages of Microservices.....12
205 3. MICROSERVICES – THREAT BACKGROUND13
206 3.1 Review of Threat Sources Landscape.....13
207 3.2 Microservices-specific Threats14
208 3.2.1 *Service Discovery Mechanism Threats*.....14
209 3.2.2 *Botnet Attacks*.....14
210 3.2.3 *Cascading Failure*14
211 4. SECURITY STRATEGIES FOR IMPLEMENTATION OF CORE FEATURES AND COUNTERING
212 THREATS.....15
213 4.1 Strategies for Identity and Access Management15
214 4.2 Strategies for Service Discovery Mechanism16
215 4.4 Strategies for Security Monitoring19
216 4.5 Availability/Resiliency Improvement Strategies.....19
217 4.5.1 *Analysis of Circuit Breaker implementation options*19
218 4.5.2 *Strategies for Load Balancing*20
219 4.5.3 *Rate Limiting (Throttling)*20
220 4.6 Integrity Assurance strategies.....21
221 4.7 Countering Botnet Attacks.....22
222 5. SECURITY STRATEGIES FOR ARCHITECTURAL FRAMEWORKS IN MICROSERVICES.....23

223 Appendix A: Differences between Monolithic Application and Microservices-based Application24

224 A.1 Design and Deployment Differences24

225 *A.1.1 An Example Application to illustrate the design and deployment differences*.....25

226 A.2 Run-time Differences26

227 Appendix B: Traceability of Security Strategies to Microservices Architectural Features28

228 Appendix C: References32

229

230

231

EXECUTIVE SUMMARY

232

233

234 The microservices paradigm is being increasingly used for designing and deploying large-scale
235 application systems in both cloud-based and enterprise infrastructures. The resulting application
236 system consists of relatively small, loosely coupled entities or components called microservices
237 that communicate with each other using lightweight communication protocols.

238

239 Incentives to design and deploy a microservices-based application system include: (a) agility in
240 development due to relatively small and less complex codebases since each one typically
241 implements a single business function; (b) independence among teams in the development
242 process thanks to the loosely coupled nature of microservices; and (c) availability of deployment
243 tools that provide infrastructure services such as authentication, access control, service discovery
244 and communication, and load balancing.

245

246 Despite several facilitating technologies (e.g., orchestration), there are many challenges to be
247 addressed in the development and deployment of a microservices-based application. Network
248 security, reliability, and latency are critical factors since every transaction implemented using
249 this type of system will involve the transmission of messages across a network. Further, the
250 presence of multiple microservices exposes a large attack surface.

251

252 The goal of this document is to outline strategies for the secure deployment of a microservices-
253 based application by analyzing the implementation options for core state of practice features,
254 considering the configuration options for architectural frameworks such as API gateway and
255 service mesh, and countering microservices-specific threats.

256

257 **1. INTRODUCTION, SCOPE, AND TARGET AUDIENCE**

258
259 Application systems are increasingly developed and deployed using the microservices paradigm
260 due to advantages such as agility, flexibility, scalability, and availability of tools for automating the
261 underlying processes. However, the tremendous increase in the number of components in a
262 microservices-based application system combined with complex network environments comprised
263 of various interaction styles among components call for several core infrastructure features to be
264 implemented either alone or bundled/packaged into architectural frameworks, such as API gateway
265 and service mesh. The objective of this document is to perform an analysis of the implementation
266 options for core features and configuration options for architectural frameworks as well as outline
267 security strategies that counter microservice-specific threats.
268

269 **1.1 Scope of this document**

270
271 This document will not discuss the various tools used in the deployment of microservices-based
272 application systems. Discussion of core features and architectural frameworks will be limited to
273 highlighting issues relevant to secure implementation. The core focus is on the methodology to
274 develop security strategies for microservices-based applications through the following three
275 fundamental steps:
276

- 277 • Study of the technology behind microservices-based application systems focusing on design
278 principles, basic building blocks, and associated infrastructure
- 279 • Focused review of the threat background specific to the operating environment of
280 microservices
- 281 • Analysis of implementation options related to state of practice core features and
282 configuration options related to architectural frameworks for developing security strategies
283

284 **1.2 Target Audience**

285
286 The target audience for the security strategies discussed in this document includes:
287

- 288 • Chief Security Officer (CSO) or Chief Technology Officer (CTO) of an IT department in a
289 private enterprise or government agency who wishes to develop enterprise infrastructures to
290 host distributed systems based on microservices architecture
- 291 • Application architects who wishes to design a microservices-based application system
292

293 **1.3 Relationship to other NIST Guidance Documents**

294
295 This is guidance document focuses on a class of application based on a specific architecture.
296 However, since an essential architectural component—the microservice—can be implemented
297 inside a container, the security guidance and recommendations related to application container
298 technology may also be relevant security strategies for the application architecture discussed in this
299 document. Such guidance includes:

- 300
301
302
303 • NIST SP 800-190, Application Container Security Guide
304 • NIST IR 8176, Security Assurance Requirements for Linux Application Container
305 Deployments
306

307 **1.4 Methodology and Organization**

308
309 Since microservices-based application systems encompass diverse technologies (e.g., server
310 virtualization, containers, cloud middleware), the focus here is on core features of this application
311 class and the architectural frameworks that bundle or package them. The threat analysis approach
312 involves taking a macro view of the entire deployment stack of microservices-based application
313 systems and the layer at which these core features are located. The threats specific to those features
314 are identified, and the overall approach for developing security strategies is to analyze the multiple
315 implementations for core features and the architectural frameworks as well as ensure that those
316 implementation options counter microservices-specific threats. The roadmap for the materials used
317 in this methodology is as follows:

- 318
319 • Review of all state of practice core features that form the infrastructure for microservices
320 (Section 2.6)
321 • Review of the layers in the deployment stack, location of the core features in those layers, and
322 identification of microservices-specific threats (Section 3)
323 • Analysis of all different implementation options for these core features and outline of security
324 strategies based on these implementation options for core features (Section 4)
325 • Review of all architectural frameworks that bundle several core features as a single product and
326 outline security strategies based on the configuration options for architectural frameworks
327 (Section 5)
328

329 A slightly more detailed summarization of the contents of the various sections in this document is
330 as follows:

- 331
332 • Chapter 2 provides a high-level but expansive overview of microservices-based application
333 systems, starting with a conceptual view followed by design principles, business drivers,
334 building blocks, component interaction styles, state of practice core features, and architectural
335 frameworks
336 • Chapter 3 provides a stack level view of the threat background and some threats that are
337 specific to the microservices environment
338 • Chapter 4 contains analysis information pertaining to various state of practice core features for
339 supporting a microservices-based application and outlines the security strategies for
340 implementing the core features based on analysis of implementation options
341 • Chapter 5 contains analysis information pertaining to architectural frameworks that bundle core
342 features needed in the infrastructure for microservices-based applications and outlines the
343 security strategies for configuring the architectural frameworks
344

2. MICROSERVICES-BASED APPLICATION SYSTEMS – TECHNOLOGY BACKGROUND

In this section, the technology behind the development and deployment of a microservices-based application system will be described using the underlying design drivers or principles, the artifacts that constitute the building blocks, and the different ways the building blocks can be configured to produce different architectural options. This is not meant to be a comprehensive description of the technology but rather provide sufficient information about components and concepts to facilitate the identification of security threats and the development of secure implementation strategies for a microservices-based application system.

2.1 Microservices – a Conceptual View

A microservices-based application system consists of multiple components (microservices) that communicate with each other through synchronous remote procedure calls or an asynchronous messaging system. Each microservice typically implements one (rarely more) distinct business process or functionality (e.g., storing customer details, storing and displaying product catalog, customer order processing). Each microservice is a mini-application that has its own business logic and various adapters for carrying out functions such as database access and messaging. Some microservices would expose a RESTful API [1] that is consumed by other microservices or by the application's clients [2]. Other microservices might implement a web UI. At runtime, a microservice instance may be configured to run as a process in an application server, in a virtual machine (VM), or in a container.

Though a microservices-based application can be implemented purely as an enterprise application and not as a cloud service, its development is often identified as cloud-native application development with a service-based architecture, application programming interface (API)-driven communications, container-based infrastructure, and a bias for DevOps processes such as continuous improvement, agile development, continuous delivery, and collaborative development among developers, quality assurance teams, security professionals, IT operations, and line-of-business stakeholders [3]. Part of the reason for this perspective is due to the fact that on-premises software development and deployment relies on a server-centric infrastructure with tightly integrated application modules rather than on loosely coupled, services-based architectures with API-based communications.

2.2 Microservices – Design Principles

The design of a microservice is based on the following drivers [4]:

- Each microservice must be managed, replicated, scaled, upgraded, and deployed independently of other microservices
- Each microservice must have a single function and operate in a bounded context (i.e., have limited responsibility and dependence on other services)

- 388 • All microservices should be designed for constant failure and recovery and must therefore be as
389 stateless as possible
- 390 • One should reuse existing trusted services (e.g., databases, caches, directories) for state
391 management

392 These drivers, in turn, result in the following design principles for a microservice:

- 393
- 394 • Autonomy
- 395 • Loose coupling
- 396 • Re-use
- 397 • Composability
- 398 • Fault tolerance
- 399 • Discoverability
- 400 • APIs alignment with business processes

401 **2.3 Business drivers**

402

403 Though the business drivers for deployment of microservices-based application systems are only
404 marginally related to the theme of this document, it is useful to identify and state those that are
405 relevant from the point of view of user and organizational behavior [5]:

- 406
- 407 • Ubiquitous access: users want access to applications from multiple client devices (e.g.,
408 browsers, mobile devices)
- 409 • Scalability: applications must be highly scalable to maintain availability in the face of
410 increasing number of users and/or increased rate of usage from the existing user base
- 411 • Agile development: organizations want frequent updates to quickly respond to
412 organizational (process and structural) changes and market demands
- 413

414 **2.4 Building Blocks**

415

416 Microservices-based applications (e.g., distributed enterprise or web applications [1]) are built
417 using an architectural style or design pattern that is not restricted to any specific technology and is
418 comprised of small independent entities (end points) that communicate with each other using
419 lightweight mechanisms. These end points are implemented using well-defined APIs. There are
420 several types of API endpoints, such as SOAP or REST (HTTP protocol). Each of the small
421 independent entities provides a distinct business capability called a “service” and may have its own
422 data store or repository. Access to these services is provided by various platforms or client types,
423 such as web browsers or mobile devices, using a component called the “client.” Together, the
424 component services and the client form the complete microservices-based application system. The
425 services in such a system may be classified as:

- 426
- 427 • Application-functionality services

- 428 • Infrastructure services (called “core features” in this document) implemented alone or
429 bundled into architectural frameworks (e.g., API gateway, service mesh), including
430 authentication and authorization, service registration and discovery, and security monitoring

431 In a microservices-based application system, each of the multiple, collaborative services can be
432 built using different technologies. This promotes the concept of technical heterogeneity, which
433 means that each service in a microservices-based application system may be written in a different
434 programming language, development platform, or using different data storage technologies. This
435 concept enables developers to choose the right tool or language depending on the type of service.
436 Thus, in a single microservices-based application system, the constituting services may be built
437 using different languages (e.g., Ruby, Golang, Java) and may be hosting different stores (e.g.,
438 document datastore, graphical DB, or multimedia DB). Each component service is developed by a
439 team—a microservice or DevOps team—which provides all of the development and operational
440 requirements for that service with a high degree of autonomy regarding development and
441 deployment techniques so long as the service functionality or service contract is agreed upon [6].
442

443 Services in microservices are separately deployed on different nodes. The communication between
444 them is transformed from a local function call to a remote call, which would affect system
445 performance due to a high latency of network communication. Thus, a lightweight communication
446 infrastructure is required.
447

448 Scaling can be applied selectively on those services that have performance bottlenecks due to
449 insufficient CPU or memory resources, while other services can continue to be run using smaller,
450 less expensive hardware. The functionality associated with such a service may be consumed in
451 different ways for different purposes, thereby promoting reusability and composability. One
452 example includes a customer database service, the contents of which are used both by shipping
453 departments for preparing bills of lading and by accounts receivable or the billing department to
454 send invoices.
455

456 **2.5 Microservices – Interaction Styles**

457

458 In monolithic applications, each component (i.e., a procedure or function) invokes another using a
459 language-level call, such as a method or function. In microservices-based applications, each service
460 is typically a process running in its own distinct network node that communicates with other
461 services through an inter-process communication mechanism (IPC) [7]. Additionally, a service is
462 defined using an interface definition language (IDL) (e.g., Swagger), resulting in an artifact called
463 the application programming interface (API). The first step in the development of a service
464 involves writing the interface definition, which is reviewed with client developers and iterated
465 multiple times before the implementation of the service begins. Thus, an API serves as a contract
466 between clients and services.
467

468 The choice of the IPC mechanism dictates the nature of the API [7]. The following table provides
469 the nature of API definitions for each IPC mechanism.
470
471

472

Table 1: IPC Mechanisms and API Types

IPC Mechanism	Nature of API Definition
Asynchronous, message-based (e.g., AMQP or STOMP)	Made up of message channels and message types
Synchronous request/response (e.g., HTTP-based REST or Thrift)	Made up of URLs and request and response formats

473

474 There can be different types of message formats used in IPC communication: text-based and
 475 human-readable, such as JSON or XML, or of a purely machine-readable binary format, such as
 476 Apache Avro or Protocol buffers.

477

478 The principle of autonomy described earlier may call for each microservice to be a self-contained
 479 entity that delivers all of the functions of an application stack. However, for a microservices-based
 480 application that provides multiple business process capabilities (e.g., an online shopping application
 481 that provides business processes such as ordering, shipping, and invoicing), a component
 482 microservice is always dependent, in some fashion, on another microservice (e.g., data). In the
 483 context of our example, the shipping microservice is dependent upon “unfulfilled orders” data in
 484 the ordering microservice to perform its function of generating a shipping or bill of lading record.
 485 Hence, there is always the need to couple microservices while still retaining autonomy. The various
 486 approaches to creating the coupling, which are often dictated by business process and IT
 487 infrastructure needs, include interaction patterns, messaging patterns, and consumption modes. In
 488 this document, the term “interaction pattern” is used, and the primary interaction patterns are as
 489 follows.

490

491 **Request-reply:** Two distinct types of requests include queries for the retrieval of information and
 492 commands for a state-changing business function [2]. In the first type, a microservice makes a
 493 specific request for information or to take some action and functionally waits for a response. The
 494 purpose of the request for information is retrieval for presentation purposes. In the second type, one
 495 microservice asks another to take some action involving a state-changing business function (e.g., a
 496 customer modifying their personal profile or submitting an order). In the request-reply pattern,
 497 there is a strong runtime dependency between the two microservices involved, which manifests in
 498 the following two ways:

499

- 500 • One microservice can execute its function only when the other microservice is available
- 501 • The microservice making the request must ensure that the request has been successfully
 502 delivered to the target microservice

503 Because of the nature of communication in the request-reply protocol, a synchronous
 504 communication protocol, such as HTTP, is used. If the microservice is implemented with a REST
 505 API, the messages between the microservices become HTTP REST API calls. The REST APIs are
 506 often defined using a standardized language, such as RAML (RESTful API Modeling Language),
 507 which was developed for microservice interface definition and publication. HTTP is a blocking
 508 type of communication wherein the client that initiates a request can continue its task only when it
 509 receives a response.

510

511 **Publish-Subscribe:** This pattern is used when microservices need to collaborate for the realization
 512 of a complex business process or transaction. This is also called a business domain event-driven

513 approach or domain event subscription approach. In this pattern, a microservices registers itself or
514 subscribes to business domain events (e.g., interested in specific information or being able to
515 handle certain requests), which are published to a message broker through an event-bus interface.
516 These microservices are built using event-driven APIs and use asynchronous messaging protocols,
517 such as Message Queuing Telemetry Transport (MQTT), Advanced Message Queuing Protocol
518 (AMQP), and Kafka Messaging, which enable support for notifications and subscriptions. In
519 asynchronous protocols, the message sender does not typically wait for a response but simply sends
520 the message to a message agent (e.g., RabbitMQ queue). One of the use cases for this approach is
521 the propagation of data updates to multiple microservices based on certain events [8].
522
523

524 **2.6 Microservices – State of the Practice Core Features**

525
526 The criticality of the communication infrastructure in a microservices-based application
527 environment calls for several sophisticated capabilities to be provided as core features in many
528 deployments. As already stated, many of these features can be implemented either stand-alone or
529 bundled together in architectural frameworks such as API gateway or service mesh. Even within
530 the API gateway, these features can be implemented through service composition or direct
531 implementation within the code base. These features include but are not limited to authentication,
532 access control, service discovery, load balancing, response caching, application-aware health
533 checks, and monitoring [2]. A brief description of these features [5] includes:
534

- 535 • Authentication and access control – The infrastructure platform can be leveraged to centralize
536 enforcement of authentication and access control for all downstream microservices, eliminating
537 the need to provide authentication and access control for each of the individual services.
538 Authentication and access policy may vary depending on the type of APIs exposed by
539 microservices—some may be public APIs; some may be private APIs; and some may be partner
540 APIs, which are available only for business partners.
- 541 • Service Discovery – In legacy distributed systems, there are multiple services configured to
542 operate at designated locations (IP address and port number). In the microservices-based
543 application, the following scenario exists and calls for a robust service discovery mechanism:
544 (a) There are a substantial number of services and many instances associated with each service
545 with dynamically changing locations.
546 (b) Each of the microservices may be implemented in VMs or containers, which may be
547 assigned dynamic IP addresses, especially when they are hosted in an IAAS or SAAS cloud
548 service.
549 (c) The number of instances associated with a service can vary based on the load using features
550 such as autoscaling.
- 551 • Security monitoring and analytics – To detect attacks and identify factors for degradation of
552 services (which may impact availability), it is necessary to monitor network traffic into and out
553 of microservices with analytics capabilities in addition to routine logging features.
554

555 An API gateway is generally needed for implementing the following core features:
556

- 557 • Optimized endpoint – This involves several capabilities.

- 558 a) *Request and response collapsing*: Most business transactions will involve calls to multiple
559 microservices, often in a pre-determined sequence. An API gateway can simplify the
560 situation for clients by exposing an endpoint that will automatically make all the needed
561 multiple requests (calls) and return a single, aggregated response to the client.
- 562 b) *API Transformation*: The API gateway can provide a public interface to the client which is
563 different from the individual APIs it has to consume or the program calls it has to make to
564 cater to a given request. This feature is called API transformation and enables:
- 565 i) Changing the implementation and even the API interface for individual microservices
566 ii) Transitioning from an initial, monolithic application to a microservices-based
567 application by enabling continued access to clients through the API gateway while
568 progressively splitting the monolithic application, creating microservice APIs in the
569 background, and changing the API transformation configuration accordingly
- 570 c) *Protocol Translation*: Calls from clients to microservices endpoints may be in web
571 protocols, such as HTTPS, while microservices communicate among themselves using
572 synchronous protocols, such as RPC/Thrift, or asynchronous protocols, such as AMQP. The
573 necessary protocol translation in client requests is typically carried out by the API gateway.
- 574 • **Circuit breaker** – This is a feature to set a threshold for the failed responses to an instance of a
575 microservice and cut off proxying requests to that instance when the failure is above the
576 threshold. This avoids the possibility of a cascaded failure, allows time to analyze logs,
577 implement the necessary fix, and push an update for the failing instance.
 - 578 • **Load balancing**: There is a need to have multiple instances of the same service, and the load on
579 these instances must be evenly distributed to avoid delayed responses or service crashes due to
580 overload.
 - 581 • **Rate limiting (throttling)** – The rate of requests coming into a service must be limited to ensured
582 continued availability of service for all clients.
 - 583 • **Blue/green deployments** – When a new version of a microservice is deployed, requests from
584 customers using the old version can be redirected to the new version since the API gateway can
585 be programmed to be aware of the locations of both versions.
 - 586 • **Canary releases** – Only a limited amount of traffic is initially sent to a new version of a
587 microservice since the correctness of its response or performance metric under all operating
588 scenarios is not fully known. Once sufficient data is gathered about its operating characteristics,
589 then all of the requests can be proxied to the new version of the microservice.

590

591 **2.7 Microservices – Architectural Frameworks**

592

593 The two main architectural frameworks for bundling or packaging core features that primarily
594 ensure reliable, resilient, and secure communication in a microservices-based application are:

595

- 596 • API gateway, augmented with or without micro gateways
- 597 • Service mesh

598

599 The role of these frameworks in the operating environment of a microservices-based application
600 system are given in Table 2 below [4]:

601
602
603

Table 2: Role of Architectural Frameworks in Microservices Operations

Architectural Framework	Role in the Overall Architecture
API gateway, augmented with or without micro gateways	Used for controlling north-south and east-west traffic (the latter using micro gateways); micro gateways are deployed when microservices are implemented in web/application servers
Service mesh	Deployed for purely east-west traffic when microservices are implemented using containers but can also be used in situations where microservices are housed in VMs or application servers

604

605 **2.7.1 API Gateway**

606

607 The API gateway is a popular architectural framework for microservices-based application systems.
 608 Unlike a monolithic application where the endpoint may be a single server, a microservices-based
 609 application consists of multiple fine-grained endpoints. Direct communication of clients to multiple
 610 endpoints results in too many point-to-point connections. Hence, it makes sense to provide a single
 611 entry point for all clients to multiple component microservices of the application. This is the
 612 underlying objective behind the API gateway architecture. The primary function of the API
 613 gateway is to support clients with different form factors (e.g., browser, mobile device) and
 614 functional requirements. The core features of the API gateway are request routing, composition,
 615 and protocol translation (i.e., translation between web protocols, such as HTTP and WebSocket,
 616 and web-unfriendly protocols that are used internally, such as AMQP and Thrift binary RPC). All
 617 requests from clients first go through the API gateway, which then routes requests to the
 618 appropriate microservice. The API gateway will often handle a request by invoking multiple
 619 microservices and aggregating the results.

620

621 The multiple APIs or microservices accessible through the API gateway can be specified as part of
 622 the input port definition of the gateway (e.g., mobileAPI or MobileService) or be specified
 623 dynamically through a deploy operation of the API gateway service with a request parameter that
 624 contains the name of the service that should be embedded with the requested service [9]. Thus, the
 625 API gateway, located between clients and microservices, represents a pattern wherein a proxy
 626 aggregates multiple services. Many API gateway implementations can support APIs written in
 627 different languages, such as Jolie, JavaScript, or Java.

628

629 Since the API gateway is the entry point for microservices, it should be equipped with the
 630 necessary infrastructure services (in addition to its main service of request shaping), such as service
 631 discovery, authentication and access control, load balancing, caching, providing custom APIs for
 632 each type of client, application-aware health checks, service monitoring, and circuit breakers. These
 633 additional features may be implemented in the API gateway in two ways:

634

- 635 • By composing the specific services developed for respective functionality (e.g., service
636 registry for service discovery)
- 637 • Implementing these functionalities directly inside the codebase that utilizes the API
638 gateway

639 **Gateway implementations**

640 To prevent the gateway from having too much logic to handle request shaping for different client
641 types, it is divided into multiple gateways [8]. This results in a pattern called backends for
642 frontends (BFF). In BFF, each client type is given its own gateway (e.g., web app BFF, mobile app
643 BFF) as a collection point for service requests. The respective backend is closely aligned with the
644 corresponding front end (client) and is typically developed by the same team.

645
646 API management for a microservices-based application can be implemented through either a
647 monolithic API gateway architecture or a distributed API gateway architecture. In the monolithic
648 API gateway architecture, there is only one API gateway that is typically deployed at the edge of
649 the enterprise network (e.g., DMZ) and provides all services to the API at the enterprise level. In
650 the distributed API gateway architecture, there are multiple instances of microgateways, which are
651 deployed closer to microservice APIs [10]. A microgateway is typically a low footprint, scriptable
652 API gateway that can be used to define and enforce customized policies and is therefore suitable for
653 microservices-based applications, which must be protected through service-specific security
654 policies.

655
656 The microgateway is typically implemented as a stand-alone container using development
657 platforms such as Node.js. It is different from a sidecar proxy of the service mesh architecture
658 (refer to Section 2.7.2), which is implemented at the API endpoint itself. The security policies in a
659 microgateway are encoded using JSON format and input through a graphical policy management
660 interface. The microgateway should contain policies for both application requests and responses.
661 Since policies and their enforcement are implemented as a container, they are immutable and thus
662 provide a degree of protection against accidental and unintended modifications that may result in
663 security breaches or conflicts, since any security policy update requires redeployment of the
664 microgateway. It is essential that the microgateway deployed for any microservice instance
665 communicate with service registry and monitoring modules to keep track of the operational status
666 of the microservice it is designed to protect.

667 **2.7.2 Service Mesh**

668
669 A service mesh is a dedicated infrastructure layer that facilitates service-to-service communication
670 through service discovery, routing and internal load balancing, traffic configuration, encryption,
671 authentication and authorization, metrics, and monitoring. It provides the capability to
672 declaratively define network behavior, node identity, and traffic flow through policy in an
673 environment of changing network topology due to service instances coming and going offline and
674 continuously being relocated. It can be looked upon as a networking model that sits at a layer of
675 abstraction above the transport layer of the OSI model (e.g., TCP/IP) and addresses the service's
676 session layer (Layer 5 of the OSI model) concerns, eliminating the need to address them through
677 application code [11]. A service mesh conceptually has two modules—the data plane and the
678 control plane. The data plane carries the application request traffic between service instances
679 through service-specific proxies. The control plane configures the data plane, provides a point of

680 aggregation for telemetry, and provides APIs for modifying the behavior of the network through
681 various features, such as load balancing, circuit breaking, or rate limiting.

682
683 Service meshes create a small proxy server instance for each service within a microservices
684 application. This specialized proxy car is sometimes called a “sidecar proxy” in service mesh
685 parlance [12]. The sidecar proxy forms the data plane, while the runtime operations needed for
686 enforcing security (access control, communication-related) are enabled by injecting policies (e.g.,
687 access control policies) into the sidecar proxy from the control plane. This also provides the
688 flexibility to dynamically change policies without modifying the microservices code.

689

690 **2.8 Comparison with Monolithic Architecture**

691
692 To fully compare the microservice architecture with the monolithic architecture used for all legacy
693 applications, it is necessary to compare the features of applications developed using these
694 architectural styles as well as provide an example of an application under both architectures for a
695 specific business process. A detailed discussion involving these aspects is provided in Appendix A.

696

697 **2.9 Comparison with Service-Oriented Architecture (SOA)**

698
699 The architectural style of microservices shares many similarities with service-oriented architecture
700 (SOA) due to the following common technical concepts [13]:

701

- 702 • Services – The application system provides its various functionalities through self-contained
703 entities or artifacts called services that may have other attributes such as being visible or
704 discoverable, stateless, reusable, composable, or have technological-diversity
- 705 • Interoperability – A service can call any other service using artifacts such as an enterprise
706 service bus (ESB) in the case of SOA or through a remote procedural call (RPC) across a
707 network as in the case of a microservices environment
- 708 • Loose coupling – There is minimal dependency between services such that the change in
709 one service does not require a change in another service

710 In spite of the three common technical concepts described above, technical opinion on the
711 relationship between an SOA and microservices environment falls along the following three lines
712 [13]:

713

- 714 • Microservices are a separate architectural style
- 715 • Microservices represent one SOA pattern
- 716 • Microservice is a refined SOA

717 The most prevalent opinion is that the differences between SOA and microservices do not concern
718 the architectural style except in its concrete realization, such as development or deployment
719 paradigms and technologies [2].

720 **2.10 Advantages of Microservices**

721

- 722 • For large applications, splitting the application into loosely coupled components enables
723 independence between the developer teams assigned to each component. Each team can then
724 optimize by choosing its own development platform, tools, language, middleware, and
725 hardware based on their appropriateness for the component being developed.
- 726 • Each of the components can be scaled independently. The targeted allocation of resources
727 results in maximum utilization of resources.
- 728 • If components have HTTP RESTful interfaces, implementation can be changed without
729 disruption to the overall function of the application as long as the interface remains the same.
- 730 • The relatively smaller codebase involved in each component enables the development team to
731 produce updates more quickly and provide the application with the agility to respond to changes
732 in business processes or market conditions.
- 733 • The loose coupling between the components enables containment of the outage of a
734 microservice such that the impact is restricted to that service without a domino effect on other
735 components or other parts of the application.
- 736 • When components are linked together using an asynchronous event-handling mechanism, the
737 impact of a component's outage is temporary since the required functions will automatically
738 execute when the component begins running again, thus maintaining the overall integrity of the
739 business process.
- 740 • By aligning the service definition to business capabilities (or by basing the decomposition logic
741 for the overall application functionality based on business processes or capabilities), the overall
742 architecture of the microservices-based system is aligned with the organizational structure. This
743 promotes agile response when business processes associated with an organizational unit change
744 and consequently require that associated service to be modified and deployed.

745 **2.11 Disadvantages of Microservices**

746

- 747 • Multiple components (microservices) must be monitored instead of one single application. A
748 central console is needed to obtain the status of each component and the overall state of the
749 application. Therefore, an infrastructure must be created with distributed monitoring and
750 centralized viewing capabilities.
- 751 • The presence of multiple components creates the availability problem since any component
752 may cease functioning at any time.
- 753 • A component may have to call the latest version of another component for some clients and call
754 the previous version of the same component for another set of clients (i.e., version
755 management).
- 756 • Running an integration test is more difficult since a test environment is needed wherein all
757 components must be working and communicating with each other.

758

759 **3. MICROSERVICES – THREAT BACKGROUND**

760
761 The threat background for a microservices-based application system should be treated as a
762 continuation of the technology background provided in Section 2. The following approach has been
763 adopted to review the threat background:

- 764
- 765 • Consider all layers in the deployment stack of a typical microservices-based application and
- 766 when identifying typical potential threats at each layer
- 767 • Identify the distinct set of threats exclusive to microservices-based application systems

768 **3.1 Review of Threat Sources Landscape**

769
770 Six layers are present in the deployment stack of a typical microservices-based application as
771 suggested in [13]: hardware, virtualization, cloud, communication, service/application, and
772 orchestration. This document considers these layers to be threat sources, and several of the security
773 concerns affiliated with them are described below to provide an overview of the threat background
774 in a microservices-based application. It is important to remember that many of the possible threats
775 are common to other application environments and not specific to a microservices-based
776 application environment.

- 777
- 778 • Hardware layer – Though hardware flaws, such as Meltdown and Spectre [8], have been
- 779 reported, such threats are rare. In the context of this document, hardware is assumed to be
- 780 trusted, and threats from this layer are not considered.
- 781 • Virtualization layer: In this layer, threats to microservices or hosting containers originate from
- 782 compromised hypervisors and the use of malicious or vulnerable container images and VM
- 783 images. These threats are addressed in other NIST documents and are therefore not discussed
- 784 here.
- 785 • Cloud environment – Since virtualization is the predominant technology used by cloud
- 786 providers, the same set of threats to the virtualization layer applies. Further, there are potential
- 787 threats within the networking infrastructure of the cloud provider. For example, hosting all
- 788 microservices within a single cloud provider may result in fewer network-level security controls
- 789 for inter-process communication as opposed to controls for communication between external
- 790 clients and the microservices hosted within the cloud. Security threats within a cloud
- 791 infrastructure are considered in several other NIST documents and are therefore not addressed
- 792 here.
- 793 • Communication layer – This layer is unique to microservices-based applications due to the
- 794 sheer number of microservices, adopted design paradigms (loose coupling and API
- 795 composition), and different interaction styles (synchronous or asynchronous) among them.
- 796 Many of the core features of microservices pertain to this layer, and the threats to these core
- 797 features are identified under microservices-specific threats in Section 3.2.
- 798 • Service/application layer – In this layer, threats are the results of malicious or faulty code. As
- 799 this falls under secure application development methodologies, it is outside of the scope of this
- 800 document.
- 801 • Orchestration layer – An orchestration layer may come into play if the microservices
- 802 implementation involves technologies such as containers. The threats in this layer pertain to the

803 subversion of automation or configuration features, especially related to scheduling and
804 clustering of servers, containers, or VMs hosting the services, and are therefore beyond the
805 scope of this document.

806 **3.2 Microservices-specific Threats**

807
808 Most state-of-practice core features refer to the communication layer in the deployment stack of
809 microservices-based applications. Hence, the overall security strategies for microservices-based
810 applications should involve choosing the right implementation options, identifying the architectural
811 frameworks packaging those core features, identifying microservice-specific threats, and providing
812 coverage for countering those threats in the implementation options.

813 **3.2.1 Service Discovery Mechanism Threats**

814 The basic functions in a service discovery mechanism are:

- 815
- 816 • Service registration and de-registration
 - 817 • Service discovery

818
819 The potential security threats to the service discovery mechanism include:

- 820
- 821 • Registering malicious nodes within the system, redirecting communication to them, and
822 subsequently compromising service discovery
 - 823 • Corruption of the service registry database leading to redirection of service requests to wrong
824 services and resulting in denial of services; also, redirection to malicious services resulting in
825 compromise of the entire application system

826 **3.2.2 Botnet Attacks**

827 Unlike monolithic applications, wherein calls to a functional module of the application originate
828 from a local procedure call or through a local data structure (i.e., sockets), calls to an API in a
829 microservices-based application always originate from a program, not a direct client or user
830 invocation), many of them from a remote program across the network. This exposes a
831 microservices API to a multitude of botnets, which can vary based on the type of damage it inflicts
832 (e.g., credential stuffing/abuse, takeover of accounts, page scraping, harvesting data, denial of
833 service).

834 **3.2.3 Cascading Failure**

835 The presence of multiple components in a microservices-based application enhances the probability
836 of a failure of a service. Though the components are designed to be loosely coupled from the point
837 of view of deployment, there is a logical or functional dependency since many business
838 transactions require the execution of multiple services in sequence to deliver the required outputs.
839 Therefore, if a service that is upstream in the processing logic of a business transaction fails, other
840 services that depend upon it may become unresponsive as well. This phenomenon is known as
841 cascading failure.

842 **4. SECURITY STRATEGIES FOR IMPLEMENTATION OF CORE FEATURES AND** 843 **COUNTERING THREATS**

844
845 Security strategies for the design and deployment of microservices-based application systems will
846 span the following:

847
848 Analysis of implementation options for core features:

- 849
- 850 a) Identity and access management
- 851 b) Service discovery
- 852 c) Secure communication protocols
- 853 d) Security monitoring
- 854 e) Resiliency or availability improvement techniques
- 855 f) Integrity assurance improvement techniques

856
857 Countering microservices-specific threats:

- 858
- 859 a) Threats to service discovery mechanism
- 860 b) Botnet attacks
- 861 c) Cascading failures

862
863 Note that service discovery is a core feature in microservices, and analysis of the implementation
864 options will also take into consideration threats to service discovery mechanisms. Similarly,
865 implementation options for resiliency or availability improvement will also address the counter
866 measures for cascading failures. As such, there will not be separate security strategies for these
867 items.

868 **4.1 Strategies for Identity and Access Management**

869
870 Since microservices are packaged as APIs, the initial form of authentication to microservices
871 involves the use of API keys (cryptographic). Authentication tokens encoded in SAML or through
872 OpenID connect under the OAuth 2.0 framework provide an option for enhancing security [14].
873 Additionally, a centralized architecture for provisioning and enforcement of access policies
874 governing access to all microservices is required due to the sheer number of services, the
875 implementation of services using APIs, and the need for service composition to support real-world
876 business transactions (e.g., customer order processing and shipping). A standardized, platform-
877 neutral method for conveying authorization decisions through a standardized token (e.g., JSON
878 web tokens (JWT), which are OAuth 2.0 access tokens encoded in JSON format [15]) is also
879 required since each of the microservices may be implemented in a different language or platform
880 framework. Policy provisioning and computation of access decisions require the use of an
881 authorization server.

882
883 The disadvantage to implementing access control policies at the access point of each microservice
884 is that additional effort is required to ensure that cross-cutting (common) policies applicable to all
885 microservice APIs are implemented uniformly. Any discrepancy in security policy implementation
886 among APIs will have security implications for the entire microservices-based application. Further,
887 the footprint for implementing access control in each microservices node can result in performance

888 issues in some nodes. Since multiple microservices nodes collaborate to perform a transaction,
889 performance problems associated with any node can quickly cascade across multiple services. The
890 strategies for secure identity and access management to microservices are outlined below.

891

892 **Security strategies for authentication (MS-SS-1):**

- 893 • Authentication to microservices APIs that have access to sensitive data should not be done
894 simply by using API keys. Rather, an additional form of authentication should also be used.
- 895 • Every API Key that is used in the application should have restrictions specified both for the
896 applications (e.g., mobile app, IP address) and the set of APIs where they can be used.

897

898 **Security strategies for access management (MS-SS-2):**

- 899 • Access policies to all APIs and their resources should be defined and provisioned centrally to
900 an access server
- 901 • The access server should be capable of supporting fine-grained policies
- 902 • Access decisions from the access server should be conveyed to individual and sets of
903 microservices through standardized tokens encoded in a platform-neutral format (e.g., OAuth
904 2.0 token encoded in JSON format)
- 905 • The scope in authorization tokens (extent of permissions and duration) should be carefully
906 controlled; for example, in a request for transaction, the allowed scope should only involve the
907 API endpoints that must be accessed for completing that transaction
- 908 • It is preferable to generate tokens for performing authentication instead of passing credentials to
909 the API endpoints since any damage would be limited to the time that the token is valid;
910 authentication tokens should be cryptographically signed or hashed tokens

911

912 **4.2 Strategies for Service Discovery Mechanism**

913

914 Microservices may have to be replicated and located anywhere in the enterprise or cloud
915 infrastructure for optimal performance and load balancing reasons. In other words, services could
916 be frequently added or removed and dynamically assigned to any network location. Hence, it is
917 inevitable in a microservices-based application architecture to have a service discovery mechanism,
918 which is typically implemented using the service registry. The service registry service is used by
919 microservices that are coming online to publish their locations in a process called service
920 registration and is also used by microservices seeking to discover registered services. The service
921 registry must therefore be configured with confidentiality, integrity, and availability
922 considerations.

923

924 In service-oriented architectures (SOA), service discovery is implemented as part of the
925 centralized enterprise service bus (ESB). However, in microservices architecture—where the
926 business functions are packaged and deployed as services within containers and communicate
927 with each other using API calls—it is necessary to implement a lightweight message bus that can
928 implement all three interaction styles mentioned in Section 2.5. Additionally, alternatives to the
929 ways in which service registry service can be implemented span two dimensions: (a) the way
930 clients access the service registry service and (b) centralized versus distributed service registry.
931 Clients can access the service registry service using two primary methods: client-side discovery
932 pattern and server-side discovery pattern [16].

933
934
935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978

Analysis of the client-side service discovery pattern

The client-side option consists of building registry-aware clients. The client queries the service registry for the location of all services needed to make requests. It then contacts the target service directly. Though simple, this implementation option for service discovery requires the discovery logic (querying the service registry) to be implemented for each programming language and/or framework that is used for client implementations.

Analysis of the service-side service discovery pattern

The service-side discovery has two implementations: one pattern delegates the discovery logic to a dedicated router service set at a fixed location, while the other utilizes a server in front of each microservice with the functionality of a dynamic DNS-resolver. In the dedicated router option, the client makes all service requests to this dedicated router service, which in turn queries the service registry for the location of the client-requested service and forwards that request to the discovered location. This removes the tight coupling between an application service and an infrastructure service such as the service registry service. In the DNS resolver pattern, each microservice completes its own service discovery using its built-in DNS resolver to query the service registry. The DNS resolver maintains a table of available service instances and their endpoint locations (i.e., IP addresses). To keep the table up to date, the asynchronous, nonblocking DNS resolver queries the service registry regularly—perhaps every few seconds—using DNS SRV records for service discovery. Since the service discovery function through the DNS resolver runs as a background task, the endpoints (URLs) for all peer microservices are instantly available when a service instance needs to make a request [2].

A good strategy would be to use a combination of the service-side service discovery pattern and the client-side service discovery pattern [16]. The former can be used for providing access to all public APIs, while the latter can allow clients to access all cluster-internal interactions.

Centralized versus distributed service registry

In a centralized service registry implementation, all services wishing to publish their service register at a single point, and all services seeking these services use the single registry to discover them. The security disadvantage of this pattern is the single point of failure [17]. However, data consistency will not be an issue. In the decentralized service registry, there may be multiple service registry instances, and services can register with any of the instances. In the short term, the disadvantage is that there will be data inconsistency between the various service registries. Eventually, consistency among these various instances of service registry is achieved either through broadcasting from one instance to all others or by propagation from one node to all others via attached data in a process called piggybacking.

Regardless of the pattern used for service discovery, secure deployment of service discovery functions should meet the following service registry configuration requirements.

Security strategies for service registry configuration (MS-SS-3)

- Service registry capabilities should be provided through a cluster of servers with a configuration that can perform frequent replication.

- 979 • Service registry clusters should be in a dedicated network where no other application
980 service is run.
- 981 • Communication between an application service and a service registry should occur through
982 a secure communication protocol such as HTTPS or TLS.
- 983 • Service registry should have validation checks to ensure that only legitimate services are
984 performing the registration, refresh operations, and database queries to discover services.
- 985 • The bounded context and loose coupling principle for microservices should be observed for
986 the service registration/deregistration functions. In other words, the application service
987 should not have tight coupling with an infrastructure service, such as a service registry
988 service, and service self-registration/deregistration patterns should be avoided. When an
989 application service crashes or is running but unable to handle requests, its inability to
990 perform deregistration affects the integrity of the whole process. Therefore,
991 registration/deregistration of an application service should be enabled using a third-party
992 registration pattern, and the application service should be restricted to querying the service
993 registry for service location information as described under the client-side discovery pattern.
- 994 • If a third-party registration pattern is implemented, registration/deregistration should only
995 take place after a health check on the application service is performed.
- 996 • Distributed service registry should be deployed for large microservices application, and care
997 should be taken to maintain data consistency among multiple service registry instances.

998

999 **4.3 Strategies for Secure Communication Protocols**

1000

1001 Secure communication between clients and services (north-south traffic) and between services
1002 (east-west traffic) is critical for the operation of a microservices-based application. It is a good
1003 practice to build security features into infrastructure rather than application code, and several
1004 technologies have evolved with that objective.

1005

1006 However, certain strategies for security services—such as authentication or the establishment of
1007 secure connections—can be handled at the individual microservices nodes. For example, in the
1008 fabric model, each microservice instance has the capability to function as an SSL client and SSL
1009 server (i.e., each microservice is an SSL/TLS endpoint). Thus, a secure SSL/TLS connection is
1010 possible for interservice or inter-process communication from an overall application perspective.
1011 These connections can be created dynamically (i.e., before each interservice request) or be created
1012 as a keep-alive connection. In the keep-alive connection scheme, a “service A” creates a connection
1013 after a full SSL/TLS handshake—the first time an instance of that service makes a request to an
1014 instance of a “service B.” However, neither service instances terminate the connection after a
1015 response returns for that request from service B. Rather, the same connection is reused in future
1016 requests. The advantage of this scheme is that the costly overhead involved in performing the initial
1017 SSL/TLS handshake can be avoided during each request, and an existing connection can be reused
1018 for thousands of following interservice requests. Thus, a permanent secure interservice network
1019 connection is available for all instances of requests.

1020

1021 **Security strategies for secure communication (MS-SS-4)**

- 1022 • Clients should not be configured to call its target services directly but rather to point to the
1023 single gateway URL

- 1024 • Client-to-API-gateway communication should take place after mutual authentication and be
1025 encrypted (e.g., using mTLS protocol)
1026 • Frequently interacting services should create keep-alive TLS connections
1027

1028 **4.4 Strategies for Security Monitoring**

1029
1030 Compared to monitoring a monolithic application which runs in a server (or some replicas for load
1031 balancing), a microservices-based system must monitor a large number of services, each running in
1032 different servers possibly hosted on heterogeneous application platforms. Further, any meaningful
1033 transaction in the system will involve at least two or more services.
1034

1035 **Security strategies for security monitoring (MS-SS-5)**

- 1036 • An analytics engine analyzes the dependencies among the services and identifies nodes
1037 (services) and paths (network) that are bottlenecks
1038 • A central dashboard displays the status of various services and the network segments that
1039 link them

1040 **4.5 Availability/Resiliency Improvement Strategies**

1041
1042 In microservices-based applications, targeted efforts that improve the availability or resiliency of
1043 certain critical services are needed to enhance the overall security profile of the application. Some
1044 technologies that are commonly deployed include:
1045

- 1046 • Circuit breaker function
1047 • Load balancing
1048 • Rate limiting (throttling)
1049

1050 ***4.5.1 Analysis of Circuit Breaker implementation options***

1051 A common strategy for preventing or minimizing cascading failures involves the use of circuit
1052 breakers, which prohibits the delivery of data to the component (microservice) that is failing
1053 beyond a specified threshold. This is also known as the fail fast principle. Since the errant service is
1054 quickly taken offline, incidences of cascading failures are minimized while the errant component's
1055 logs are analyzed, required fixes are performed, and microservices are updated. There are three
1056 options for deploying circuit breakers [9]: directly inside the client, on the side of services, or in
1057 proxies that operate between clients and services.
1058

1059 Client-side circuit breaker option: In this option, each client has a separate circuit breaker for each
1060 external service that the client calls. When the circuit breaker in a client has decided to cut off calls
1061 to a service (called “open state” with respect to that service), no message will be sent to the service,
1062 and communication traffic in the network is subsequently reduced. Moreover, the circuit breaker
1063 functionality need not be implemented in the microservice, which frees valuable resources for
1064 efficient implementation of that service. However, locating the circuit breaker in the client carries
1065 two disadvantages from a security point of view. First, a great deal of trust must be placed in the
1066 client that the circuit breaker code executes properly. Second, the overall integrity of the operation
1067 is at risk since knowledge of the unavailability of the service is very much local to the client, a

1068 status that is determined based on the frequency of calls from that client to the service rather than
1069 on the combined response status received by all clients against that service.

1070
1071 Server-side circuit breaker option: In this option, an internal circuit breaker in the microservice
1072 processes all client invocations and decides whether it should be allowed to invoke the service or
1073 not. The security advantages of this option are that clients need not be trusted to implement the
1074 circuit breaker function, and since the service has a global picture of the frequency of all
1075 invocations from all clients, it can throttle requests to a level which it can conveniently handle (e.g.,
1076 temporarily lighten the load).

1077
1078 Proxy circuit breaker option: In this option, circuit breakers are deployed in a proxy service, located
1079 between clients and microservices, which handles all incoming and outgoing messages. Within this,
1080 there may be two options: one proxy for each target microservice or a single proxy for multiple
1081 services (usually implemented in API gateway) that includes both client-side circuit breakers and
1082 service-side circuit breakers existing within that proxy. The security advantage of this option is that
1083 neither the client code nor the services code needs to be modified, which avoids trust and integrity
1084 assurance issues associated with both these categories of code as well as the circuit breaker
1085 function. This option also provides additional protections such as making clients more resilient to
1086 faulty services, and shielding services from cases in which a single client sends too many requests
1087 [9], resulting in some type of denial of service to other clients that use that service.

1088
1089 **Security strategies for implementing circuit breakers (MS-SS-6)**
1090 • A proxy circuit breaker option should be deployed to limit the trusted component to the proxy.
1091 This avoids the need to place the trust on the clients and microservices (e.g., setting thresholds
1092 and cutting off requests based on the set threshold) since they are multiple components.

1093
1094 **4.5.2 Strategies for Load Balancing**
1095 Load balancing is an integral functional module in all microservices-based applications, and its
1096 main purpose is to distribute loads to services. A service name is associated with a namespace
1097 that supports multiple instances of the same service. In other words, many instances of the same
1098 service would use the same namespace [17]. To balance the service load, the load balancer
1099 chooses one service instance in the request namespace using an algorithm such as the round-robin
1100 algorithm—a circular pattern to assign the request to a service instance.

1101
1102 **Security strategies for load balancing (MS-SS-7)**
1103 • All programs supporting the load balancing function should be decoupled from individual
1104 service requests. For example, the program that performs health checks on services to
1105 determine the load balancing pool should run asynchronously in the background.
1106 • When a DNS resolver is deployed in front of a source microservice to provide a table of
1107 available target microservice instances, it should work in tandem with the health check program
1108 to present a single list to the calling microservice.

1109 **4.5.3 Rate Limiting (Throttling)**
1110 The goal of rate limiting is to ensure that a service is not oversubscribed. That is, when one client
1111 increases the rate of requests, the service continues its response to other clients. This is achieved by
1112 setting a limit on how often a client can call a service within a defined window of time. When the

1113 limit is exceeded, the client—rather than receiving an application-related response—receives a
1114 notification that the allowed rate has been exceeded as well as additional data regarding the limit
1115 number and the time at which the limit counter will be reset for the requestor to resume receiving
1116 responses. Closely related to the concept of rate limiting is quota management or conditional rate
1117 limiting where limits are determined based on application requirements rather than infrastructure
1118 limitations or requirements.

1119

1120 **Security strategies for rate limiting (MS-SS-8)**

- 1121 • Quotas or limits for application usage should be based on both infrastructure and application-
1122 related requirements.
- 1123 • Limits should be determined based on well-defined API usage plans.

1124

1125 **4.6 Integrity Assurance strategies**

1126

1127 Integrity assurance requirements in the context of microservices-based applications arise under two
1128 contexts:

1129

- 1130 • When new versions of microservices are inducted into the system
- 1131 • For supporting session persistence during transaction

1132

1133 Monitored induction of new releases: Whenever a newer version of a microservice is released, its
1134 induction must be a gradual process since (a) all clients may not be ready to use the new version,
1135 and (b) the behavior of the new version for all scenarios and use cases may not meet the business
1136 process expectation despite extensive testing. To address this situation, a technique called canary
1137 release is often adopted [4]. Under this technique, only a limited number of requests are routed to
1138 the new version after it is brought online, and the rest are routed to the existing operational version.
1139 After a period of observation provides assurance that the new version meets performance and
1140 integrity metrics, all of the requests are routed to the new version.

1141

1142 **Security (integrity assurance) strategies for the induction of new versions of microservices** 1143 **(MS-SS-9):**

- 1144 • The traffic to both the existing version and the new version of the service should be routed
1145 through a central node, such as an API gateway, to monitor the total number of calls to the
1146 service.
- 1147 • Security monitoring should cover nodes hosting both the existing and newer versions.
- 1148 • Usage monitoring of the existing version should steadily increase traffic to the new version.
- 1149 • The performance and functional correctness of the new version should be factors in increasing
1150 traffic to the new version.
- 1151 • Client preference for the version (existing or new) should be taken into consideration while
1152 designing a canary release technique.

1153

1154 Session persistence: It is critical to send all requests in a client session to the same upstream
1155 microservice instance since clients execute a complete transaction through multiple requests to a
1156 specific service, and the target of all requests should be to the same upstream service instance in
1157 that session. This requirement is called session persistence. A situation that could potentially break
1158 this requirement is one wherein the microservice stores its state locally, and the load balancer

1159 handling individual requests forwards a request from an in-progress user session to a different
1160 microservice server or instance. One of the methods for implementing session persistence is sticky
1161 cookie. In this method, there is a mechanism to add a session cookie to the first response from the
1162 upstream microservice group to a given client, identifying (in an encoded fashion) the server that
1163 generated the response. Subsequent requests from the client include the cookie value, and the same
1164 mechanism uses it to route the request to the same upstream server [18].
1165

1166 **Security (integrity assurance) strategies for handling session persistence (MS-SS-10):**

- 1167 • The session information for a client must be stored securely
- 1168 • The artifact used for conveying the binding server information must be protected

1169

1170 **4.7 Countering Botnet Attacks**

1171
1172 Though it is impossible to protect against all types of botnets, microservice APIs must be provided
1173 with detection and prevention capabilities against credential-stuffing and credential abuse attacks.
1174 This is especially critical for those applications where each of the microservices are independently
1175 callable and carry their own sets of credentials. Credential abuse attacks can be detected using
1176 offline threat analysis or run-time solutions [19]. Detection of botnet attacks is provided by a
1177 dedicated bot manager product or as an add-on feature in web application firewalls (WAF).
1178

1179 **Security strategies for preventing credential abuse and stuffing attacks (MS-SS-11):**

- 1180 • A run-time prevention strategy for credential abuse is preferable to offline strategy. A threshold
1181 for a designated time interval from a given location (e.g., IP address) for the number of login
1182 attempts should be established; if the threshold is exceeded, prevention measures must be
1183 triggered.
- 1184 • A credential-stuffing detection solution has the capability to check user logins against the stolen
1185 credential database and warn legitimate users that their credentials have been stolen.

5. SECURITY STRATEGIES FOR ARCHITECTURAL FRAMEWORKS IN MICROSERVICES

The two main architectural frameworks considered in this document for microservices-based application systems are the API gateway and service mesh. The primary security considerations in the implementation of the API gateway involve choosing the right platform for hosting it, proper integration and configuration with enterprise-wide authentication and authorization frameworks, and securely leveraging the traffic flowing through it for security monitoring and analysis.

Security strategy for API gateway implementation (MS-SS-12):

- API gateway platform requirements: Since some microservices have multiple communication styles (i.e., synchronous and asynchronous), it is imperative that the API gateway that serves as the entry point for these services should support multiple communication protocols, and a high-performance webserver and reverse proxy should support its basic functional capabilities.
- Integrate API gateway with an identity management application to provision credentials before activating the API.
- When identity management is invoked through the API gateway, connectors should be provided for integrating with identity providers (IdPs).
- The API gateway should have a connector to an artifact that can generate an access token for the client request (e.g., OAuth 2.0 Authorization Server).
- Securely channel all traffic information to a monitoring and/or analytics application for detecting attacks (e.g., denial of service, malicious actions) and unearthing explanations for degrading performance.

Implementing a service mesh can help ensure that proper configuration parameters associated with various security policies are defined correctly in the control plane so that the intent of the security policies are met, and the service mesh alone does not introduce new vulnerabilities.

Security strategy for service mesh implementation (MS-SS-13):

- Provide policy support for designating a specific communication protocol between pairs of services and specifying the traffic load between pairs of services based on application requirements.
- Default configuration should always enable access control policies for all services.
- Avoid configurations that may lead to privilege escalation (e.g., the service role permissions and binding of the service role to service user accounts).

1225 **Appendix A: Differences between Monolithic Application and Microservices-based** 1226 **Application**

1227 **A.1 Design and Deployment Differences**

1228
1229 Conceptually, a monolithic architecture of an application involves generating one huge artifact that
1230 must be deployed in its entirety, while a microservices-based application contains multiple self-
1231 contained, loosely-coupled executables called services or microservices. The individual services
1232 can be deployed independently. In monolithic applications, any change to a certain functionality of
1233 the overall application will involve recompilation and, in some instances, re-testing of the whole
1234 application before being deployed again. However, in the case of microservices, only the relevant
1235 service is modified and redeployed since the independent nature of the services ensures that a
1236 change in one does not logically affect the functionality of another. In monolithic applications, any
1237 increase in workload due to an increase in the number of users or the frequency of application
1238 usage will involve allocating resources to the whole application, whereas in microservices, the
1239 increase in resources can be selectively applied to those whose performance is less than desirable,
1240 thus providing flexibility in scalability efforts.

1241
1242 Some monolithic applications may be constructed modularly but may not have semantic or logical
1243 modularity. Modular construction refers to how an application may be built from a large number of
1244 components and libraries that may have been supplied by different vendors, and some components
1245 (e.g., database) may be distributed across the network [17]. In such monolithic applications, the
1246 design and specification of APIs may be similar to that in a microservices architecture. However,
1247 the difference between such modularly designed monolithic applications (sometimes called a
1248 classic modular design) and a microservices-based application is that in the latter, the individual
1249 API is network-exposed and therefore independently callable and re-usable.

1250
1251 The differences between monolithic and microservices-based applications is summarized in Table
1252 A.1 below:

1253 **Table A.1: Logical Differences between Monolithic and Microservices-based Application**
1254
1255

Monolithic Application	Microservices-based Application
Must be deployed as a whole	Independent or selective deployment of services
Change in a small part of the application requires re-deployment of the entire application	Only the modified services need to be re-deployed
Scalability involves allocating resources to the application as a whole	Each of the individual services can be selectively scaled up by allocating more resources
API calls are local	Network-exposed APIs enable independent invocation and re-usability

1256

1257 **A.1.1 An Example Application to illustrate the design and deployment differences**

1258
 1259 The following example of a small, online retail application illustrates the design and deployment
 1260 differences discussed above. The main functions of this application are:

- 1261
- 1262 • A module that displays the catalog of products offered by the retailer with pictures of the
- 1263 products, product numbers, product names, and the unit prices
- 1264 • A module for processing customer orders by gathering information about the customer (e.g.,
- 1265 name, address) and the details of the order (e.g., name of the product from the catalog,
- 1266 quantity, unit price) as well as creating a bin containing all the items ordered in that session
- 1267 • A module for preparing the order for shipping, specifying the total bill of lading (i.e., the
- 1268 total package to be shipped, quantity of each item in the order, shipping preferences,
- 1269 shipping address)
- 1270 • A module for invoicing the customer with a built-in feature for making payments by credit
- 1271 card or bank account

1272
 1273 The differences in the design of this online retail application as a monolithic versus microservices-
 1274 based are given in table below.

1275 **Table A.2: Differences in Application Construct between Monolithic and Microservices-based Application**

Application Construct	Monolith	Microservices-based
Communication between functional modules	All communications are in the form of procedure calls or some internal data structures (e.g., socket). The module handling the order processing makes a procedural call to the module handling the shipping function and waits for successful completion (blocking type synchronous communication).	The shipping functionality and the order processing functionality are each designed as independent services. Communication takes place as an API call across the network using a web protocol. The order processing microservice can either (a) make a request-response call to the shipping microservice and wait for a response or (b) put the details of the order to be shipped in a message queue to be picked up asynchronously by the shipping microservice, which has subscribed to the event.
Handling changes or enhancements (e.g., invoicing module needs to be changed to accept debit cards)	The entire application must be recompiled and redeployed after making the necessary changes.	The invoicing function is designed as a separate microservice, so that service can simply recompiled and redeployed.
Scaling the application, allocation of increased	The order processing functionality involves longer	It is enough to allocate increased resources for

resources (e.g., order processing module needs to be allocated more resources to handle a larger load)	transaction times compared to shipping or invoicing functions. Vertical scaling that involves using servers with more memory or CPUs must be deployed for the entire application.	hardware where the order processing microservice is deployed. Also, the number of instances of order-processing microservices can be increased for better load balancing.
Development and deployment strategy	Development is handled by the development team which, after necessary testing by the QA team, transfers the task of deployment to an infrastructure team that oversees the allocation of suitable resources for deployment.	The complete lifecycle—from development to deployment—is handled by a single DevOps team for each microservice since it is a relatively small module with a single functionality and built-in a platform (e.g., OS, languages libraries) that is optimal for that functionality.

1278

1279 **A.2 Run-time Differences**

1280

1281 A monolithic application runs as a single computational node such that the node is aware of the
 1282 overall system or application state. In a microservices environment, the application is designed as a
 1283 set of multiple nodes that each provide a service. Since they operate without the need to coordinate
 1284 with others, the overall system state is unknown to individual nodes. In the absence of any global
 1285 information or global variable values, the individual nodes make decisions based on locally
 1286 available information. The independence of the nodes means that failure of one node does not
 1287 affect other nodes. Unlike monolithic applications where services may share database connections
 1288 or a data repository, a microservice architecture may deploy a pattern wherein each service has its
 1289 own data repository. In many situations, interaction between services may require a distributed
 1290 transaction which, if not designed properly, may affect the integrity of the databases.

1291

1292 The runtime differences between monolithic and microservices applications and their implications
 1293 are summarized in Table A.2 below.

1294

1295 **Table A.3: Architectural Differences between Monolithic and Microservices-based Application**

1296

Monolithic Application	Microservices-based Application
Runs as a single computational node; overall state information fully known	Designed as a set of multiple nodes, each providing a service; overall system state is unknown to individual nodes
Designed to make use of global information or values of global variables	Individual nodes make decisions based on locally available information
Failure of the node means crash of the application	Failure of one node should not affect other nodes

1297

Figure A.1: Online Shopping Application – Monolithic Architecture

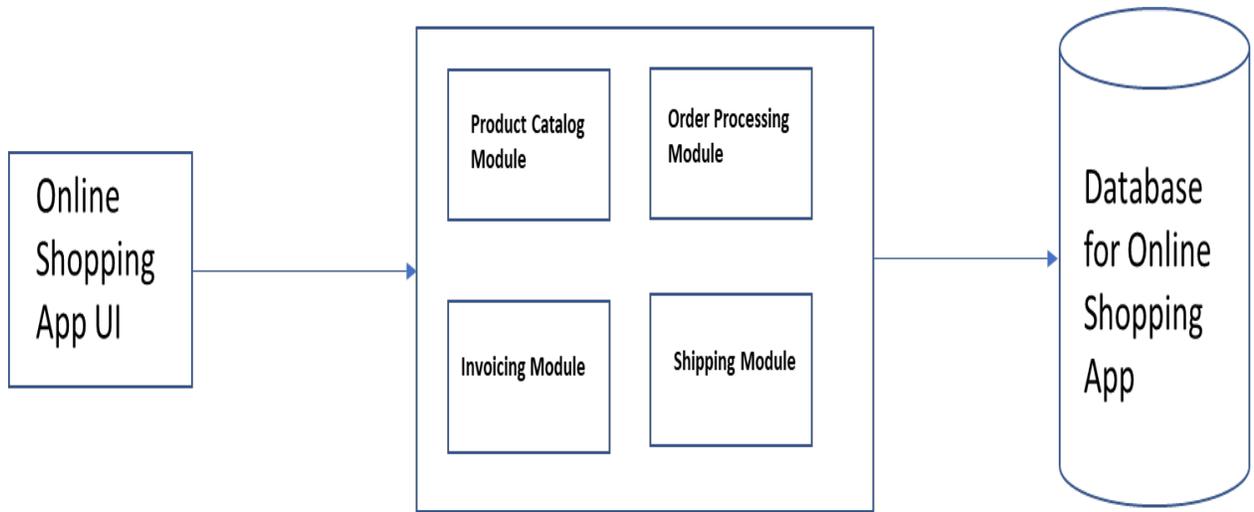


Figure A.1: Online Shopping Application – Monolithic Architecture

1298
1299
1300
1301
1302
1303
1304
1305

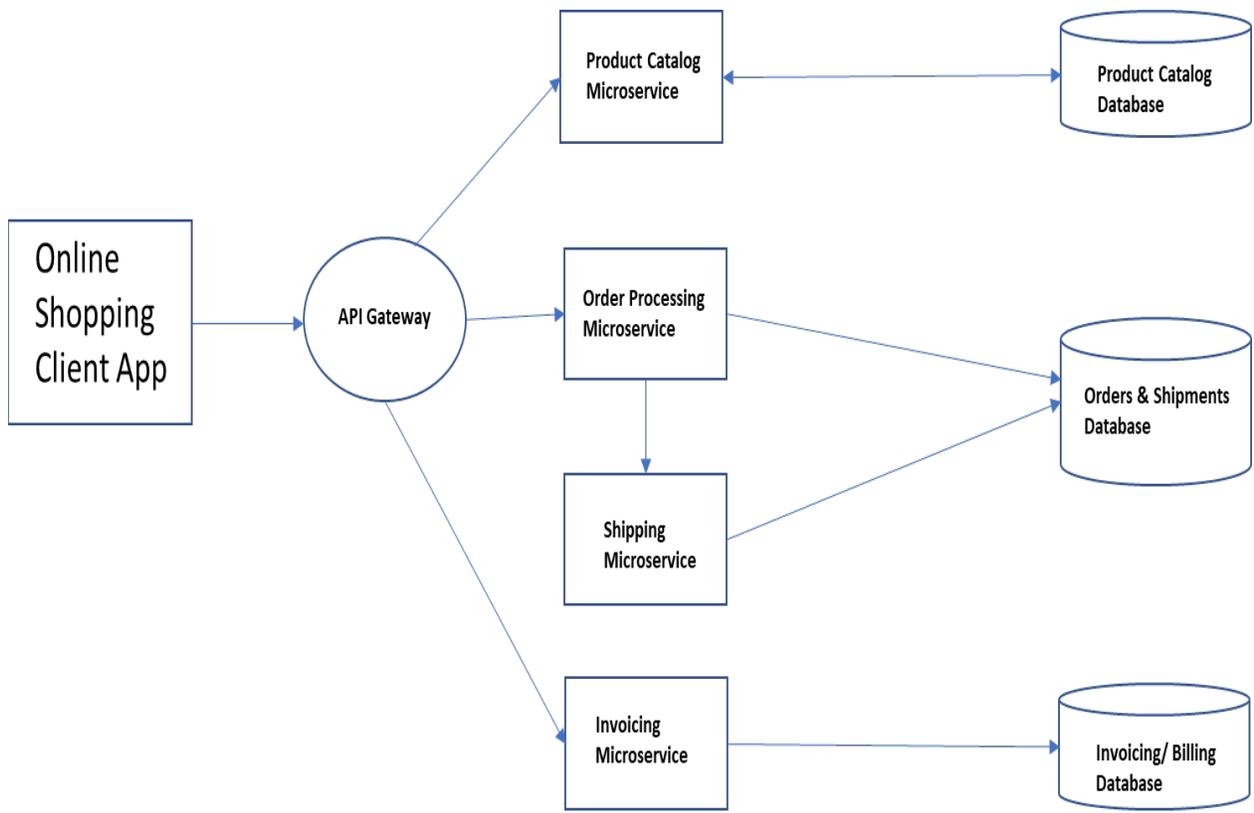


Figure A.2: Online Shopping Application – Microservices Architecture

1306
1307
1308

1309
1310
1311

Appendix B: Traceability of Security Strategies to Microservices Architectural Features

Security Strategy Identifier	Security Strategy	Microservices Core Feature/ Architectural framework
MS-SS-1	<ul style="list-style-type: none"> • Authentication to microservice APIs that have access to sensitive data should not be done simply by using API keys; an additional form of authentication should also be used. • Every API Key that is used in the application should have restrictions specified both for applications (e.g., mobile app, IP address) and the set of APIs where they can be used 	Authentication
MS-SS-2	<ul style="list-style-type: none"> • Access policies to all APIs and their resources should be defined and provisioned centrally to an access cover • The access server should be capable of supporting fine-grained policies • Access decisions from the access server should be conveyed to individual and sets of microservices through standardized tokens encoded in a platform-neutral format (e.g., OAuth 2.0 token encoded in JSON format) • The scope in authorization tokens (i.e., extent of permissions and duration) should be carefully controlled; for example, in a request for a transaction, the allowed scope should only involve the API endpoints that must be accessed to complete that transaction • It is preferable to generate tokens for performing authentication instead of passing credentials to the API endpoints since potential damage will be limited to the time that the token is valid instead of the long-term damage due to compromised credentials; authentication tokens should be cryptographically signed or hashed 	Access management

Security Strategy Identifier	Security Strategy	Microservices Core Feature/ Architectural framework
MS-SS-3	<ul style="list-style-type: none"> • Service registry capability should be provided through a cluster of servers with a configuration that can perform frequent replication • Service registry clusters should be in a dedicated network where no other application services are run • Communication between an application service and a service registry should be through a secure communication protocol, such as HTTPS/TLS • Service registry should have validation checks to ensure that only legitimate services are performing the registration and refresh operations or querying its database to discover services • The bounded context and loose coupling principle for microservices should be observed for the service registration/deregistration function; the application service should not have tight coupling with an infrastructure service, such as service registry service, and the service self-registration/deregistration pattern should be avoided. Moreover, when an application service crashes or is running but not in a position to handle requests, it cannot perform deregistration, thus affecting the integrity of the whole process. Registration or deregistration of an application service should be enabled using a third-party registration pattern, and the application service should be restricted to simply querying the service registry for service location information as described in the client-side discovery pattern. • If third-party registration pattern is implemented, registration/deregistration should only take place after performing a health check on the application service • Distributed service registry should be deployed for large microservices applications, and care should be taken to maintain data consistency among multiple service registry instances 	Service registry configuration
MS-SS-4	<ul style="list-style-type: none"> • Clients should not be configured to call their target services directly but rather be configured to point to the single gateway URL • Client to API gateway communication should take place after mutual authentication and be encrypted (e.g., using mTLS protocol) • Frequently interacting services should create keep-alive TLS connections 	Secure communication

Security Strategy Identifier	Security Strategy	Microservices Core Feature/ Architectural framework
MS-SS-5	<ul style="list-style-type: none"> • Analytics engine that analyzes dependencies among the services and identifies nodes (services) and paths (network) that are the bottlenecks • A central dashboard that displays the status of various services and the network segments linking them 	Security monitoring
MS-SS-6	<ul style="list-style-type: none"> • Proxy circuit breaker option should be deployed to limit the trusted component to be the proxy, which avoids the need to place the trust on the clients and microservices (setting thresholds and cutting off requests based on the set threshold) since they are multiple components 	Implementing circuit breaker
MS-SS-7	<ul style="list-style-type: none"> • The load balancing function should be decoupled from individual service requests; for example, the program that performs health checks on the services to determine the load balancing pool should run asynchronously in the background • When a DNS resolver is deployed in front of a source microservice to provide a table of available target microservice instances, it should work in tandem with the health check program to present a single list to the calling microservice 	Implementing load balancing
MS-SS-8	<ul style="list-style-type: none"> • Quotas or limits for application usage should be based on both infrastructure and application-related requirements • Limits should be determined based on well-defined API usage plans 	Rate limiting (throttling)
MS-SS-9	<ul style="list-style-type: none"> • Traffic to both the existing version and the new version of the service should be routed through a central node, such as an API gateway, to monitor the total number of calls to the service • Security monitoring should cover nodes hosting both the existing and newer versions 	Induction of new versions of microservice
MS-SS-10	<ul style="list-style-type: none"> • Session information for a client must be stored securely • The artifact used for conveying the binding server information must be protected 	Handling session persistence
MS-SS-11	<ul style="list-style-type: none"> • A run-time prevention strategy for credential abuse is preferable to an offline strategy; a threshold for a designated time interval from a given location (e.g., IP address) for the number of login attempts should be set up, and prevention measures must be triggered if the threshold is exceeded • A credential-stuffing detection solution with the capability to check user logins against the stolen credential database and warn the legitimate users that their credentials have been stolen 	Preventing credential abuse and stuffing attacks

Security Strategy Identifier	Security Strategy	Microservices Core Feature/ Architectural framework
MS-SS-12	<ul style="list-style-type: none"> • Channel all traffic information to a monitoring and/or analytics application for detecting attacks (e.g., denial of service, malicious threats) through unusual usage patterns or deteriorating response times • Integrate API gateway with an identity management application to provision credentials before activating the API • API gateway platform requirements: since some microservices have multiple communication styles (i.e., synchronous and asynchronous), it is imperative that the API gateway which serves as the entry point for these services support multiple communication styles; a high-performance webserver and reverse proxy should support its basic functional capabilities • When identity management is invoked through an API gateway, connectors should be provided for integrating with IdPs • API gateway should have a connector to an artifact that can generate an access token for the client request (e.g., OAuth 2.0 Authorization Server) 	API gateway configuration
MS-SS-13	<ul style="list-style-type: none"> • Policy support should be enabled for: (a) designating a specific communication protocol between pairs of services and (b) specifying the traffic load between pairs of services based on application requirements • Default configuration should always be to enable access control policies for all services • Avoid configurations that may lead to privilege escalation (e.g., the service role permissions and binding of the service role to service user accounts) 	Service mesh configuration

Appendix C: References

- 1313
1314
1315 [1] Sill, A. (2016). The design and architecture of microservices. IEEE Cloud Computing.
1316
1317 [2] Richardson, C. and Smith, F. (2016). Microservices: From design to deployment. NGINX
1318 Inc.
1319
1320 [3] TechTarget. (n.d.). Comparing two schools of application development: Traditional vs.
1321 Cloud-Native. Retrieved from
1322 [https://searchcloudcomputing.techtarget.com/PaaS/Comparing-Two-Schools-of-](https://searchcloudcomputing.techtarget.com/PaaS/Comparing-Two-Schools-of-Application-Development-Traditional-vs-Cloud-Native)
1323 [Application-Development-Traditional-vs-Cloud-Native](https://searchcloudcomputing.techtarget.com/PaaS/Comparing-Two-Schools-of-Application-Development-Traditional-vs-Cloud-Native).
1324
1325 [4] Richardson, C. (2015). Building microservices: Using an API gateway. Retrieved from
1326 <https://www.nginx.com/blog/building-microservices-using-an-api-gateway/>.
1327
1328 [5] Palladino, M. (2016). Microservices and API gateway, Part 1: Why an API gateway?
1329 Retrieved from www.nginx.conf. Accessed March 2019.
1330
1331 [6] Jander, K., Braubach, L., and Pokahr, A. (2018). Defense in-depth and role authentication
1332 for microservice systems. Proceedings of the 9th International Conference on Ambient
1333 Systems, Networks and Technologies.
1334
1335 [7] Richardson, C. (2015). Building microservices: Inter-process communication in a
1336 microservices architecture. Retrieved from [https://www.nginx.com/blog/building-](https://www.nginx.com/blog/building-microservices-inter-process-communication/)
1337 [microservices-inter-process-communication/](https://www.nginx.com/blog/building-microservices-inter-process-communication/).
1338
1339 [8] Guidelines for Adopting Frontend Architectures and Patterns in Microservices-Based
1340 Systems, Proceedings of 11th Joint Meeting of the European Software Engineering
1341 Conference (ESEC) and the ACM SIGSOFT Symposium on the Foundations of
1342 Software Engineering (FSE), Paderborn, Germany, Sept 4-8, 2017.
1343
1344 [9] Montesi, F. and Weber, J. (2016). Circuit breakers, discovery, and API gateways in
1345 microservices.
1346
1347 [10] O’Neill, M. and Malinverno, P. (2018). Critical capabilities for full life cycle API
1348 management. Gartner Report 334223.
1349
1350 [11] Calcote, L. (2018). The enterprise path to service mesh architectures. O’Reilly.
1351
1352 [12] Twistlock. (n.d.) Securing the service mesh: Understanding the value of service meshes,
1353 why Istio is rising in popularity, and exploring official Twistlock compliance checks for
1354 Istio. Retrieved from [https://www.twistlock.com/resources/securing-service-mesh-istio-](https://www.twistlock.com/resources/securing-service-mesh-istio-compliance-checks/)
1355 [compliance-checks/](https://www.twistlock.com/resources/securing-service-mesh-istio-compliance-checks/).
1356
1357 [13] Yarygina, T. and Bagge, A.H. (2018). Overcoming security challenges in microservice
1358 architecture. Proceedings of 2018 IEEE Symposium on Service-Oriented System
1359 Engineering.

- 1360
1361 [14] OpenID. (n.d.). Welcome to OpenID Connect. Retrieved from <https://openid.net/connect/>.
1362 Accessed March 2019.
1363
- 1364 [15] Internet Engineering Task Force (IETF). (2012). The OAuth 2.0 authorization framework.
1365 Retrieved from <https://tools.ietf.org/html/rfc6749>. Accessed March 2019.
1366
- 1367 [16] Circuit Breakers, Discovery, and API Gateways in Microservices
1368
- 1369 [17] T. Yarygina and A.H. Bagge, “Overcoming Security Challenges in Microservice
1370 Architecture”, Proceedings of 2018 IEEE Symposium on Service-Oriented System
1371 Engineering, 2018.
1372
- 1373 [18] NGINX. (n.d.). High-performance load balancing: Scale out your applications with
1374 NGINX and NGINX Plus. Retrieved from [https://www.nginx.com/products/nginx/load-
1375 balancing/#session-persistence](https://www.nginx.com/products/nginx/load-balancing/#session-persistence).
1376
- 1377 [19] Katz, O. (2017). Improving credential abuse threat mitigation. Retrieved from
1378 <https://blogs.akamai.com/2017/01/improving-credential-abuse-threat-mitigation.html>.
1379 Accessed March 2019.