Code Unlocked

Unveiling the Future of Computing:

A Comprehensive Guide to Modern Technologies

Jithu Morrison S

PUBLISHED BY ME

WWW.JITHUMORRISON.GITHUB.IO

Dive into the ever-evolving world of computer science. This book covers the fundamentals and innovations shaping the tech landscape, from artificial intelligence and data science to software development and cybersecurity. Designed for learners and professionals alike, it offers clear explanations, real-world applications, and insights into emerging trends, equipping you to thrive in the digital age.

First printing, March 2013

Contents

Programming Languages

1. Programming Languages Overview

1.1 What is Programming Language?

Programming languages are tools that developers use to communicate with computers. Each language has its own syntax, rules, and paradigms that determine how programs are written and executed.

1.2 Types of programming language

1. Procedural Programming
2. Object-Oriented Programming
3. Functional Programming
4. Scripting Languages
5. Domain-Specific Languages

1.3 Procedural Programming

Focuses on procedures or routines (e.g., C, Python).

1.3.1 C

- Overview: One of the oldest and most influential programming languages, C is a general-purpose language that provides low-level access to memory and system processes.
- Use Cases: Systems programming (operating systems, embedded systems), game development, compilers.
- Strengths: Fast execution, fine-grained control over hardware, portability.
- Paradigm: Procedural programming.
- Notable Features: Pointers, manual memory management, close-to-hardware programming.

1.3.2 Go (Golang)

- Overview: Go is an open-source programming language designed for system-level programming and efficient concurrency, created by Google.
- Use Cases: Cloud infrastructure, server-side programming, microservices, web servers, CLI tools.
- Strengths: Concurrency model (goroutines), fast execution, simple syntax, scalability.
- Paradigm: Procedural programming with strong support for concurrency.
- Notable Features: Goroutines, fast compilation, garbage collection, static typing.

1.4 Object-Oriented Programming (OOP)

Focuses on objects (data structures) and their interactions (e.g., Java, C++, C#).

1.4.1 Java

- Overview: Java is an object-oriented, high-level programming language known for its "write once, run anywhere" capability, thanks to the Java Virtual Machine (JVM).
- Use Cases: Enterprise applications, mobile apps (Android), web servers, cloud-based systems.
- Strengths: Platform independence, large ecosystem, strong memory management (garbage collection).
- Paradigm: Object-oriented programming.
- Notable Features: Platform-independent bytecode, multithreading, strong typing, automatic garbage collection.

1.4.2 C#

- Overview: Developed by Microsoft, C# is a modern, object-oriented programming language commonly used in game development and building Windows applications.
- Use Cases: Game development (Unity), desktop applications, enterprise software, mobile apps (Xamarin).
- Strengths: Modern language features (LINQ, async/await), integration with Microsoft technologies, versatile.
- Paradigm: Object-oriented programming.
- Notable Features: Automatic garbage collection, strong typing, multithreading, event-driven programming.

1.4.3 Ruby

- Overview: Ruby is a dynamic, interpreted, high-level language known for its simplicity and productivity, particularly popular in web development via Ruby on Rails.
- Use Cases: Web development, scripting, automation, DevOps (Chef).
- Strengths: Elegant syntax, fast development, dynamic typing, highly supportive community.
- Paradigm: Object-oriented programming.
- Notable Features: Everything is an object, flexible and dynamic, built-in metaprogramming.

1.4.4 Dart

- Overview: Dart is an open-source language developed by Google, known for its use in Flutter to create cross-platform mobile applications.
- Use Cases: Mobile app development (Flutter), web apps, server-side programming.
- Strengths: Efficient cross-platform app development, reactive programming (streams), fast.
- Paradigm: Object-oriented and reactive programming
- Notable Features: Just-in-time (JIT) and ahead-of-time (AOT) compilation, strong typing, async/await support.

1.5 Functional Programming

Treats computation as the evaluation of mathematical functions, avoiding changing-state and mutable data (e.g., Haskell, Scala, Lisp).

1.5.1 Haskell

- Overview: Haskell is a purely functional programming language with strong static typing and lazy evaluation. It is known for its expressive type system and mathematical precision.
- Use Cases: Academic research, financial systems, compilers, concurrent and distributed systems.
- Strengths: Pure functions, immutability, strong typing, concise and expressive code.
- Paradigm: Purely functional programming.
- Notable Features: Lazy evaluation, Type inference, Immutability, Monads

1.6 Declarative Programming

Focuses on what the program should accomplish rather than how (e.g., SQL, Prolog).

1.6.1 SQL

- Overview: SQL (Structured Query Language) is a domain-specific language used for managing and querying relational databases.
- Use Cases: Database management, data analysis, back-end development.
- Strengths: Easy to learn, efficient for querying large datasets, standardized.
- Paradigm: Declarative programming.
- Notable Features: Queries (SELECT, INSERT, UPDATE), transactions, joins, indexing.

1.7 Hybrid Programming

Focuses on procedures or routines (e.g., C, Python).

1.7.1 C++

- Overview: C++ is an extension of C that incorporates object-oriented programming features while retaining the efficiency of C.
- Use Cases: Game engines, high-performance applications, desktop applications, systems programming.
- Strengths: Supports both low-level and high-level programming, efficient performance, large libraries.
- Paradigm: Multi-paradigm (supports procedural, object-oriented, and generic programming).
- Notable Features: Classes, inheritance, polymorphism, templates, RAII (Resource Acquisition Is Initialization).

1.7.2 Python

- Overview: Python is a versatile, high-level programming language known for its simplicity and readability, making it popular among beginners and experts alike.
- Use Cases: Web development (Django, Flask), data science, AI/ML, scripting, automation.
- Strengths: Simple and clean syntax, vast libraries, cross-platform support, fast development cycles.
- Paradigm: Multi-paradigm (supports procedural, object-oriented, and functional programming).
- Notable Features: Interpreted, dynamic typing, extensive standard libraries (e.g., NumPy, TensorFlow), easy integration with other languages (C, C++).

1.7.3 JavaScript

- Overview: JavaScript is the scripting language of the web, primarily used for interactive front-end development but also popular in backend development through Node.js.
- Use Cases: Web applications, server-side programming (Node.js), mobile apps (React Native), game development.
- Strengths: Ubiquity in browsers, asynchronous capabilities (event-driven), large community and libraries.
- Paradigm: Multi-paradigm (supports procedural, object-oriented, and functional programming).
- Notable Features: First-class functions, closures, event-driven programming, lightweight execution.

1.7.4 Swift

- Overview: Swift is a powerful, intuitive language created by Apple for iOS, macOS, watchOS, and tvOS development.
- Use Cases: iOS and macOS applications, Apple Watch and Apple TV apps.
- Strengths: Fast performance, safety features (null safety, error handling), modern syntax.
- Paradigm: Multi-paradigm (supports object-oriented and functional programming).
- Notable Features: Optional types, type inference, closures, protocol-oriented programming.

1.7.5 Kotlin

- Overview: Kotlin is a modern programming language designed to fully interoperate with Java, and it has become the preferred language for Android development.
- Use Cases: Android apps, server-side applications, full-stack web development.
- Strengths: Concise syntax, null safety, full Java compatibility, good for modern Android app development.
- Paradigm: Multi-paradigm (supports object-oriented and functional programming).
- Notable Features: Null safety, extension functions, coroutines for asynchronous programming.

1.7.6 PHP

- Overview: PHP is a server-side scripting language designed for web development but also used as a general-purpose programming language.
- Use Cases: Dynamic web pages, content management systems (CMS), e-commerce platforms (WordPress, Drupal).
- Strengths: Easy integration with HTML, large ecosystem for web development, good for rapid development.
- Paradigm: Multi-paradigm (procedural, object-oriented).
- Notable Features: Server-side execution, simple syntax, strong support for databases.

1.7.7 Rust

- Overview: Rust is a systems programming language focused on safety and performance, particularly in the area of memory management.
- Use Cases: Systems programming, embedded systems, web assembly, game engines.
- Strengths: Memory safety without a garbage collector, concurrency without data races, high performance.
- Paradigm: Multi-paradigm (supports functional and imperative programming).
- Notable Features: Ownership model, zero-cost abstractions, safe concurrency, pattern matching.

Database Systems

2. Database

2.1 Database System

A database system is a structured collection of data stored and managed in a way that allows efficient access, retrieval, modification, and management. It consists of two main components: the database (the collection of data) and the Database Management System (DBMS) (the software that handles the database).

2.2 Key Concepts

- Database: A database is an organized collection of data. It stores data in tables or structures such as records, objects, or key-value pairs, allowing efficient access and management.
- DBMS: A Database Management System (DBMS) is software that provides an interface for users and applications to interact with the database. It handles the organization, storage, retrieval, and updating of data.
- Data Models: Databases use specific models to define how data is stored, accessed, and manipulated. The most common models are:

2.2.1 Data Models

- Relational Model: Data is organized into tables (relations) and accessed using Structured Query Language (SQL). Each table consists of rows and columns. Most popular databases (e.g., MySQL, PostgreSQL) use this model.
- NoSQL Model: Non-relational databases that store data in various formats such as key-value pairs, documents, wide-column stores, or graphs. These are used for scalability and performance, particularly in big data or distributed systems.
- Hierarchical Model: Data is organized in a tree-like structure, with parent-child relationships. Used less frequently today.
- Network Model: Similar to the hierarchical model but allows more complex relationships through graph structures. This model is used in graph databases.

2.3 Types of Databases

- Relational Databases (RDBMS)
- NoSQL Databases
- Cloud Databases

- In-Memory Databases
- Distributed Databases
- Graph Databases

2.3.1 Components of DBMS

- Database Engine: Core component that handles the storage, retrieval, and management of data. It also ensures that all data is stored securely and efficiently.
- Query Processor: Interprets and executes database queries (usually written in SQL). It optimizes query performance by choosing the best way to access data.
- Transaction Manager: Manages the execution of transactions, ensuring that the database maintains consistency, especially during concurrent access and potential failures.
- Concurrency Control: Manages access to the database when multiple users are interacting with it simultaneously, ensuring data integrity.
- Backup and Recovery: Ensures that the database can recover from system failures by maintaining backups and recovery mechanisms.
- Security Manager: Controls access to the database, enforcing policies to protect data through authentication, authorization, and encryption.

2.4 Transactions

A transaction is a sequence of operations performed as a single logical unit of work. For example, transferring money from one bank account to another requires multiple operations, but the entire process is treated as one transaction. ACID Properties ensure the reliability of transactions

2.4.1 ACID

1. Atomicity: All operations within a transaction are treated as a single unit. Either all operations complete, or none do.
2. Consistency: The database must always transition from one valid state to another, maintaining integrity constraints.
3. Isolation: Transactions are executed independently, ensuring that they do not interfere with each other.
4. Durability: Once a transaction is committed, its effects are permanent, even in the event of a system crash.

2.5 SQL (Structured Query Language)

SQL is the standard language used for managing and manipulating relational databases.

2.5.1 SQL commands

1. Data Definition Language (DDL): Defines the database schema (structure).
   (a) CREATE: Create tables or databases.
   (b) ALTER: Modify the structure of tables.
   (c) DROP: Delete tables or databases.
2. Data Manipulation Language (DML): Manipulates the data stored in the database.
   (a) SELECT: Retrieve data from the database.
   (b) INSERT: Add new data to the database.
   (c) UPDATE: Modify existing data.
   (d) DELETE: Remove data from the database.
3. Data Control Language (DCL): Manages user permissions.
   (a) GRANT: Give permissions to users.
   (b) REVOKE: Remove permissions from users.
4. Transaction Control Language (TCL): Manages transactions within the database.
   (a) COMMIT: Save the changes made by a transaction.
   (b) ROLLBACK: Undo the changes made by a transaction.

2.6 Database Normalization

Normalization is the process of organizing data in a database to reduce redundancy and improve data integrity. This is done by dividing larger tables into smaller, more manageable tables and defining relationships between them.

1. 1NF (First Normal Form): Eliminate duplicate columns and ensure each field contains only atomic values.
2. 2NF (Second Normal Form): Remove subsets of data that apply to multiple rows and place them in separate tables.
3. 3NF (Third Normal Form): Remove columns that are not dependent on the primary key.

2.7 Indexing

Indexes are data structures that improve the speed of data retrieval operations on a database table by providing quick lookups. While they speed up SELECT queries, they can slow down INSERT and UPDATE operations due to the overhead of maintaining the index.

2.8 Backup and Recovery

Backup and recovery mechanisms ensure that database data can be restored in the event of data loss, system crashes, or corruption. Databases support full, incremental, and differential backups, depending on the frequency and data protection needs.

2.9 Scaling Databases

1. Vertical Scaling (Scaling Up): Adding more resources (CPU, RAM, etc.) to a single server. Limited by the capacity of the hardware.
2. Horizontal Scaling (Scaling Out): Adding more servers to distribute the load. Often used with distributed databases to handle large amounts of data and traffic.

3. Relational Database

3.1 Relational Database (RDBMS)

A Relational Database Management System (RDBMS) is a type of database management system that stores data in tables, which are composed of rows and columns. Each table represents a relation, and relationships between tables are established through primary keys and foreign keys. Data in an RDBMS is managed using Structured Query Language (SQL), a standard language for interacting with relational databases.

1. Table: A table (relation) is a collection of data organized in rows and columns. Each row represents a record, and each column represents an attribute.
2. Primary Key: A column (or set of columns) that uniquely identifies each record in a table.
3. Foreign Key: A column (or set of columns) that creates a link between two tables. The foreign key in one table refers to the primary key in another table.

3.1.1 Oracle Database

Oracle is one of the most powerful and feature-rich RDBMSs. It's widely used in large enterprises for mission-critical applications due to its scalability, performance, and security. Use Cases: Large-scale enterprise systems, complex applications with high transaction volumes, and organizations requiring extensive customization and optimization.

3.1.2 MySQL

MySQL is an open-source RDBMS, widely known for its ease of use, speed, and flexibility. It is often used in web applications, especially in combination with PHP (LAMP stack). Use Cases: Web applications, content management systems, data warehouses, and e-commerce platforms.

3.1.3 PostgreSQL

PostgreSQL is a highly advanced open-source RDBMS with strong support for SQL compliance, extensibility, and a rich feature set. It is known for its reliability and data integrity. Use Cases: Financial systems, data analysis, GIS applications, and multi-purpose enterprise applications.

3.1.4 Microsoft SQL Server

SQL Server is Microsoft's enterprise-grade RDBMS with a strong integration with the Microsoft ecosystem. It is known for its ease of use, security, and built-in business intelligence tools. Use Cases: Business-critical applications, data warehousing, business intelligence, and enterprise applications in Windows environments.

3.1.5 SQLite

SQLite is a lightweight, file-based RDBMS that is embedded within applications. It is serverless and designed to be self-contained, making it ideal for smaller applications. Use Cases: Mobile apps, IoT devices, small-scale desktop applications, prototyping, and file-based applications.

3.1.6 MariaDB

MariaDB is a fork of MySQL, created by the original MySQL developers. It is fully compatible with MySQL but offers additional features and enhanced performance. Use Cases: Web applications, data warehousing, and distributed systems.

3.1.7 IBM Db2

IBM Db2 is a robust and scalable RDBMS designed for large enterprise workloads. It offers advanced features for big data, analytics, and AI integration. Use Cases: Large enterprises, analytics-driven applications, and hybrid cloud environments.

3.1.8 Amazon Aurora

Amazon Aurora is a cloud-native RDBMS developed by AWS. It is fully compatible with MySQL and PostgreSQL but offers performance improvements and scalability for cloud-based applications. Use Cases: Cloud-native applications, web-scale applications, and SaaS platforms.

3.1.9 SAP HANA

SAP HANA is an in-memory RDBMS designed for real-time analytics and high-speed transactional processing. It is widely used in large-scale ERP (Enterprise Resource Planning) systems. Use Cases: Enterprise resource planning (ERP), supply chain management, real-time analytics, and big data applications.

3.1.10 CockroachDB

CockroachDB is a distributed SQL database designed for high availability, horizontal scaling, and strong consistency, making it ideal for cloud-native applications. Use Cases: Distributed cloud applications, high-availability systems, and globally distributed data architectures.

3.1.11 Firebird

Firebird is an open-source RDBMS known for its lightweight footprint, making it suitable for embedded systems as well as enterprise environments. Use Cases: Embedded systems, enterprise applications with moderate data processing needs.

3.1.12 Teradata

Teradata is an enterprise RDBMS known for handling large-scale data warehouses. It is designed for analytics and data mining. Use Cases: Large-scale data warehouses, real-time analytics, business intelligence, and machine learning.

3.1.13 Snowflake

Snowflake is a cloud-native data warehousing solution that uses a unique architecture to decouple storage and compute, offering scalability and flexibility for analytics. Use Cases: Cloud-based data warehousing, real-time analytics, big data processing.

3.2 SQL (Structured Query Language) in RDBMS

SQL is the standard language used to interact with relational databases. SQL commands are classified into different types based on their functionality.

III

Operating Systems

4. Operating Systems Overview

Operating Systems manage hardware and software. They act as a bridge between users and machines.

4.1 Purpose of an OS

- **Resource Management:** Allocates and deallocates CPU, memory, storage, and I/O devices.
- **Process Management:** Creates, schedules, and terminates processes.
- **File System Management:** Manages files and directories on storage devices.
- **Device Management:** Controls access to peripheral devices like printers and disks.
- **Security:** Protects the system from unauthorized access and threats.
- **User Interface:** Provides a way for users to interact with the system.

4.2 Types of Operating Systems

- **Batch OS:** Executes jobs in groups without user interaction.
- **Time-Sharing OS:** Allows multiple users to share resources simultaneously.
- **Distributed OS:** Coordinates independent computers to act as one.
- **Real-Time OS:** Ensures timely processing for critical tasks.
- **Embedded OS:** Designed for devices like smartphones and IoT systems.
- **Network OS:** Manages networking and data sharing between systems.
- **Mobile OS:** Optimized for mobile devices with touch interfaces.

4.3 Functions of an OS

- **Process Scheduling:** Decides the order in which processes execute.
- **Memory Management:** Tracks memory usage and allocation.
- **Storage Management:** Handles data storage and retrieval efficiently.
- **Multitasking:** Allows multiple programs to run concurrently.
- **Error Handling:** Detects and resolves hardware and software errors.

4.4 Examples of Operating Systems

- **Windows:** Widely used for personal computers.

- **Linux:** Open-source and highly customizable.
- **macOS:** Known for its stability and user-friendly interface.
- **Android:** Popular for mobile devices.
- **iOS:** Optimized for Apple devices.

5. Resource Management in OS

Resource management is a core function of an OS, ensuring efficient utilization of system resources. The OS manages hardware and software resources to meet user and system demands.

5.1 CPU Management

- **Process Scheduling:** Determines the execution order of processes using scheduling algorithms (e.g., FCFS, Round Robin, Priority Scheduling).
- **Context Switching:** Saves and restores the state of processes to enable multitasking.
- **Load Balancing:** Distributes CPU load evenly across all cores to maximize performance.
- **Interrupt Handling:** Responds to hardware and software interrupts efficiently.

5.2 Memory Management

- **Memory Allocation:** Allocates memory blocks to processes as needed.
- **Virtual Memory:** Extends physical memory using disk space, enabling larger applications to run.
- **Paging:** Divides memory into fixed-size pages for efficient allocation.
- **Segmentation:** Divides memory into variable-sized segments based on logical divisions (e.g., code, data).
- **Garbage Collection:** Frees unused memory to avoid memory leaks.

5.3 Storage Management

- **File Allocation:** Allocates storage space for files using methods like contiguous, linked, and indexed allocation.
- **Disk Scheduling:** Optimizes the order of I/O requests to reduce seek time (e.g., FCFS, SSTF).
- **Caching:** Stores frequently accessed data in faster storage for quicker retrieval.
- **File System Management:** Organizes files into hierarchical structures and enforces access permissions.

5.4 I/O Device Management

- **Device Drivers:** Provides a communication layer between the OS and hardware devices.

- **Device Scheduling:** Manages access to shared devices to avoid conflicts.
- **Buffering:** Temporarily stores data during I/O operations to smooth differences in speed between devices.
- **Spooling:** Manages devices like printers by queuing tasks for sequential execution.

5.5 Network Resource Management

- **Bandwidth Allocation:** Manages the distribution of network bandwidth among users and applications.
- **Connection Management:** Handles the setup, maintenance, and termination of network connections.
- **Protocol Handling:** Implements communication protocols (e.g., TCP/IP, HTTP).
- **Security:** Enforces firewalls, encryption, and user authentication for secure data transfer.

5.6 Power Management

- **Power Scheduling:** Adjusts CPU, disk, and display activity to save energy.
- **Sleep Modes:** Puts the system in low-power states when inactive.
- **Battery Monitoring:** Manages power usage in portable devices like laptops and smartphones.

5.7 Resource Allocation and Deadlock Management

- **Resource Allocation:** Ensures fair and efficient allocation of resources like CPU, memory, and devices.
- **Deadlock Prevention:** Implements strategies (e.g., avoiding circular wait) to prevent resource deadlocks.
- **Deadlock Detection:** Identifies deadlocks and resolves them by terminating or restarting processes.
- **Priority Management:** Allocates resources based on process priority.

5.8 Performance Monitoring

- **Logging and Auditing:** Tracks resource usage for optimization and troubleshooting.
- **Performance Analysis:** Identifies bottlenecks in resource utilization.
- **Load Monitoring:** Detects system overload and redistributes resources as needed.

IV

Github

6. GitHub: A Collaborative Development Platform

GitHub is a web-based platform for version control and collaboration, built on Git. It enables developers to manage code repositories, track changes, and collaborate on projects effectively.

6.1 Key Features of GitHub

6.1.1 Version Control

- Tracks changes in code over time.
- Allows users to revert to previous versions if needed.
- Facilitates branching and merging for feature development.

6.1.2 Repositories

- Centralized storage for code, documentation, and other project files.
- Can be public (open to all) or private (restricted access).
- Supports cloning for local development.

6.1.3 Collaboration

- Enables multiple developers to work on the same project.
- Pull requests allow for code review and discussion.
- Issues help track bugs, feature requests, and project tasks.

6.1.4 Branching and Merging

- Branching allows independent development without affecting the main code.
- Merging integrates changes from different branches into the main branch.
- Supports resolving conflicts during merge operations.

6.1.5 GitHub Actions

- Automates workflows like testing, building, and deployment.
- Supports Continuous Integration/Continuous Deployment (CI/CD).
- Allows custom scripts to run on specific events (e.g., push, pull request).

6.1.6 Project Management

- Includes Kanban-style boards for tracking tasks.
- Milestones group issues into specific goals or phases.
- Labels categorize issues and pull requests for better organization.

6.1.7 Code Hosting and Sharing

- Facilitates sharing projects with collaborators or the public.
- Supports binary files, images, and markdown documentation.
- Provides easy integration with third-party tools.

6.1.8 Community Building

- Enables collaboration through forks and pull requests.
- Supports discussions and wikis for project documentation.
- Encourages open-source contributions to public repositories.

6.1.9 Security and Access Control

- Offers role-based access controls for teams and collaborators.
- Scans code for vulnerabilities using Dependabot.
- Allows branch protection to enforce review policies.

6.1.10 GitHub Pages

- Hosts static websites directly from repositories.
- Supports custom domains and Jekyll-based static site generation.
- Ideal for documentation, portfolios, and project showcases.

6.1.11 Integration and API Support

- Integrates with tools like Slack, Trello, and Jenkins.
- Provides APIs for custom application development.
- Supports webhooks to trigger actions on external systems.

6.2 Conclusion

GitHub simplifies collaborative development and project management with its extensive features. It is widely used for both open-source and enterprise projects, making it a cornerstone of modern software development.

7. Github Command Usage

7.1 Github clone a Website

1. Create a repo in github or select a already created rep
2. Copy the link in the repo
3. git clone https://github.com/JithuMorrison/repo_name

7.2 Github commands to execute to push a file in cloud repo first time

1. Create a folder in pc with files eg.) create a folder using vite@latest webname - React.js
2. Create a repo in github
3. Go into the folder
4. git init
5. git remote add origin https://github.com/JithuMorrison/cloud_repo_name
6. git add .
7. git commit -m "first commit"
8. git branch
9. git branch -m main
10. git push -u origin main

7.3 Github commands to execute to push a file in cloud repo second or above

1. git add .
2. git commit -m "commit message"
3. git push -u origin main

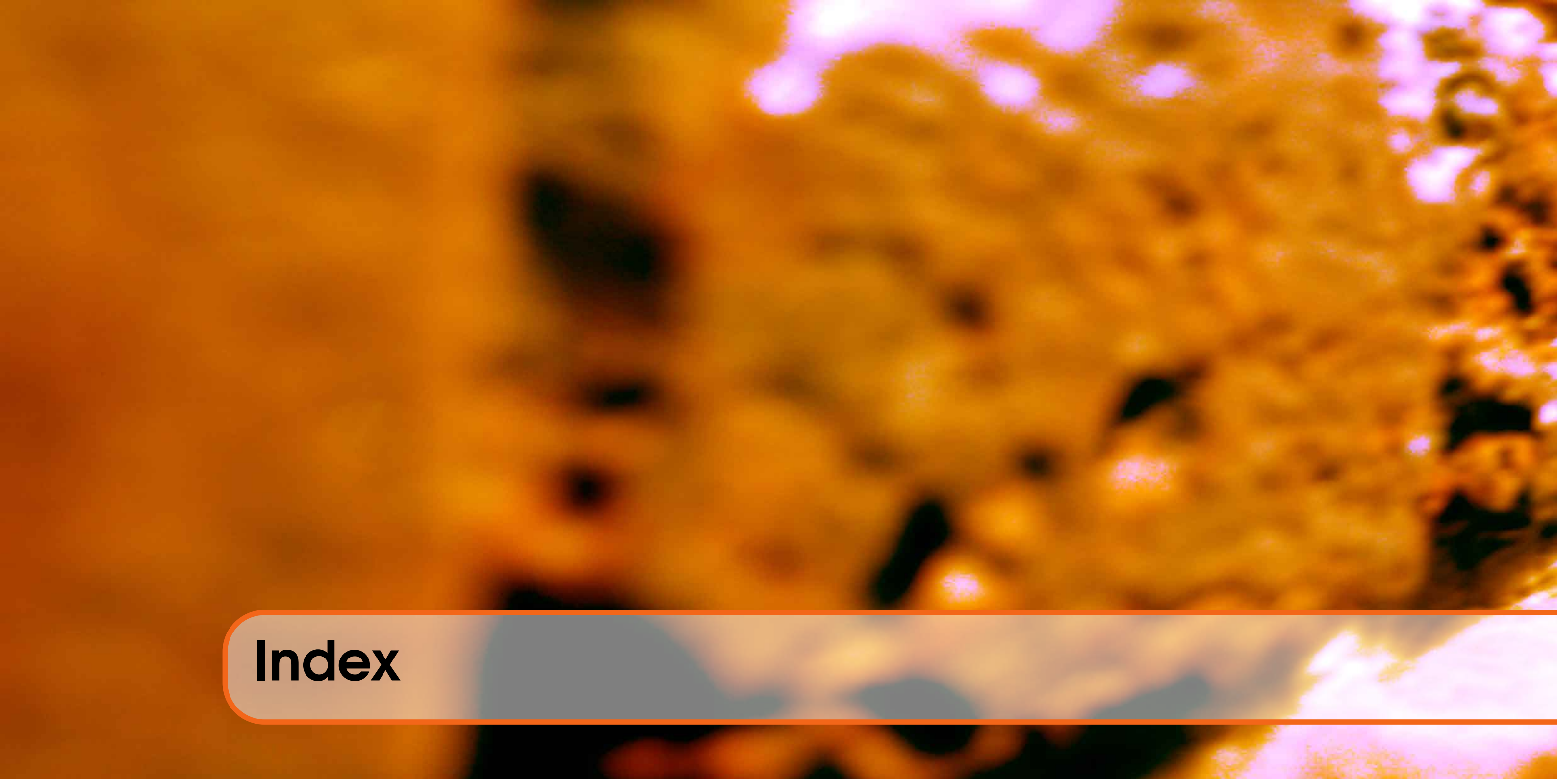
7.4 Github commands to deploy react code for first time

1. git init [No need again if already done once]
2. git remote add origin https://github.com/JithuMorrison/cloud_repo_name [Only once]
3. npm install –save gh-pages –> [In terminal]
4. "homepage": "https://jithumorrison.github.io/repo_name", –> [Fisrt line in package.json after "{"]

5. "scripts": {
   "predeploy": "npm run build",
   "deploy": "gh-pages -d build"
   } –> [In package.json]
6. export default defineConfig({
   plugins: [react()],
   base: '/repo_name/', –Start with slash
   }); –> [In vite.config.js]
7. npm run build
8. mv dist build
9. npm run deploy

7.5 Github commands to deploy react code

1. delete previous build
2. npm run build
3. mv dist build
4. npm run deploy



Index