COMP2511 — Week 9 Tutorial: Suggested Solutions, Explanations

[Confidential – Not for student release]

The suggested solutions below are provided for reference only. Students are not expected to produce answers with this level of detail. As long as their responses address the key points, they are acceptable. Please also keep in mind that alternative answers are possible, if you consider them correct, marks should be awarded accordingly.

Keep in mind that it's easy to obtain answers using generative AI ☺; therefore, simply presenting an answer is not sufficient. Emphasis in this tutorial/lab should be on active in-person interaction, encouraging students to explain their reasoning and provide constructive feedback on solutions from other students. You are encouraged to ask follow-up questions or slightly modify the problem/context to assess whether students genuinely understand what they are writing.

1. What are architectural characteristics in the context of software architecture? Provide three examples of how these characteristics can directly influence architectural decisions or software design.

Architectural characteristics, also known as non-functional requirements or quality attributes, describe how well a software system performs its functions. They define its operational behaviour, development constraints, and overall quality, dictating its fitness for purpose beyond core functionalities.

Examples of Influence on Architectural Decisions and Software Design:

1. Security

Definition: System's ability to protect data and operations from unauthorized access or attacks.

Suitable Architectures:

- **Layered**:
  - Natural separation of concerns (e.g., presentation, business, data)
  - Easier to centralize security checks (e.g., auth in middleware)
- **Microservices**:
  - Fine-grained access control per service
  - Scoped attack surface if well designed

Less Suitable Architectures:

- **Monolithic**:
  - Broader attack surface due to tight coupling
  - A single breach can expose the whole system
- **Event-Driven**:
  - Harder to track end-to-end flow

    - Security challenges in ensuring message integrity, authentication across producers/consumers

2. Performance

Definition: System speed and responsiveness under load.

Suitable Architectures:

- **Event-Driven**
    - High throughput via async, parallel processing
    - Non-blocking operations
- **Microservices**
    - Optimize & scale services independently
    - Minimize performance bottlenecks
- **Monolithic**
    - Fast for simple use cases (in-process calls)
    - Can degrade under high load due to scaling limits

Less Suitable Architecture:

- **Layered**
    - Layer interactions can add latency
    - Optimization may break abstraction principles
    - Scales like a monolith

3. Scalability

Definition: System's capacity to handle growth in workload or users.

Suitable Architectures:

- **Microservices**
    - Independent horizontal scaling per service
    - Efficient resource use
- **Event-Driven**
    - Workload distribution via queues
    - Easy horizontal scaling of consumers

Less Suitable Architectures:

- **Monolithic**
    - Scaling replicates full app
    - Inefficient; vertical scaling has limits
- **Layered**
    - Full-stack replication needed for scaling
    - Same issues as monolith in single-unit deployment

2. Why are architectural characteristics considered as important as, or even more important than, functional requirements when designing a software system?

A system that meets its functional goals but fails in core quality attributes (like scalability, performance, or security) can still *fail* in practice.

Example 1 – Online Ticketing System (Performance/Scalability)

- **Functional Success:**
  - Users can browse events, select seats, and pay.
- **Architectural Failure:**
  - Poor scalability may result in system crashes during high demand (e.g., concert sales).
  - Result: Thousands locked out, lost sales, public backlash.
- **Lesson:**
  - Performance and scalability are *mission-critical* in high-traffic applications.

Example 2 – Healthcare Patient Data System (Security)

- **Functional Success:**
  - Doctors manage patient records accurately.
- **Architectural Failure:**
  - Inadequate security (e.g., weak access control, no encryption).
  - Result: Major data breach may result in legal penalties, patient harm, trust loss.
- **Lesson:**
  - Security is *non-negotiable* in sensitive domains like healthcare.

In summary: Architectural characteristics define *how well* a system performs under real-world conditions. Even perfect functionality becomes irrelevant if the system is insecure, unstable, or unscalable.

3. Beyond direct stakeholder input, what are some other common sources or business drivers that often reveal critical architectural characteristics?

Architectural drivers often emerge from broader business and technical contexts, not just direct stakeholder requests.

1. Organizational Strategic Goals

- Drive characteristics like:
  - **Time to market**, **modifiability**, **scalability**
- Example: A company aiming for rapid global expansion needs systems that scale quickly and adapt to new regions.

2. Regulatory Compliance & Legal Requirements

- Enforce characteristics such as:
  - **Security**, **privacy**, **auditability**
- Example: Healthcare and finance systems must meet strict data protection laws.

3. Operational Environment & Existing Infrastructure

- Influence:
    - **Interoperability**, **deployability**, **operability**
- Example: Integration with legacy systems may limit technology choices or deployment strategies.

4. Market Trends & Competitive Pressure

- Push for:
    - **Performance**, **availability**, **evolvability**
- Example: A streaming service must offer low-latency playback to stay competitive.

Summary: Understanding architectural characteristics requires looking beyond user stories — strategic, legal, technical, and market forces all shape the *"how well"* of a system.

4. What are trade-offs in architecture? Give an example.

Improving one architectural characteristic often weakens another, architects must balance based on priorities.

Example: Performance vs. Security:

- High Security: encryption, logging = slower response time
- High Performance: may bypass intensive security checks
- Must weigh based on system needs (e.g., finance app prioritizes security)

5. Why document architectural characteristics and their prioritisation?

- **Shared Understanding** – Aligns stakeholders on quality goals
- **Guides Design Choices** – Helps resolve trade-offs
- **Supports Testing** – Enables performance/security validation
- **Manages Expectations** – Clarifies scope and rationale
- **Mitigates Risks** – Exposes gaps early
- **Eases Maintenance** – Aids future developers
- **Justifies Decisions** – Provides record for accountability

6. Communication: Monolith vs. Microservices

Monolith:

- In-process calls (e.g., function/method calls)
- Synchronous, tightly coupled
- Fast, simple
- Hard to isolate failures or scale parts independently

Microservices:

- HTTP/REST (sync) or messaging (async)
- Loosely coupled, over network
- Independent scaling, fault isolation
- More complex: latency, tracing, retries, versioning

7. Risks of Event-Driven Architecture (EDA) in critical systems

- **Eventual Consistency** – Not suitable for instant consistency needs
- **Error Handling** – Tracing issues is hard, errors may be silent
- **Operational Overhead** – Complex deployment & monitoring
- **Event Storms & Failures** – Overloaded queues, poison messages
- **Governance Gaps** – Schema mismatches, data duplication

8. When are Microservices a poor choice?

Scenario: Small, stable app with a small team and low growth

Why Not Microservices:

- Adds unnecessary complexity (tooling, ops, communication)
- Higher infra and maintenance cost
- Monolith is simpler, cheaper, easier to manage for such cases

9. Why use a Modular Monolith before Microservices?

- **Less Complex** – Single deployment, fewer moving parts
- **Improved Structure** – Enforces module boundaries
- **Easier Migration** – Modules map cleanly to future services
- **Skill Growth** – Lets teams adopt microservice thinking gradually
- **Cost-Effective** – Avoids early microservice overhead

10. Best architecture for strong consistency?

Monolithic or Layered (single deployment)

- Centralized DB = strong consistency
- Supports ACID transactions
- Simpler to manage integrity (e.g., financial systems)

Why not others:

- Microservices: need complex distributed coordination
- Event-Driven: async & eventual consistency by nature

11. Benefits & Challenges of Four Patterns

Layered Architecture

- **Benefit:** Clear structure, easy maintenance
- **Challenge:** Latency from layer hopping, internal coupling

Monolithic Architecture

- **Benefit:** Simple to build & deploy, fast in-process calls
- **Challenge:** Hard to scale or modify as it grows

Microservices Architecture

- **Benefit:** Independent scaling, tech flexibility
- **Challenge:** Complex to manage (distributed ops, data consistency)

Event-Driven Architecture

- **Benefit:** High scalability, loosely coupled
- **Challenge:** Debugging/tracing is hard, eventual consistency only

---

Discussion Points

1. Is it better to optimize early for scalability or wait until it becomes a problem?

It's generally **better to design for scalability early but optimize only when it becomes a problem**. This approach is known as "design for scalability, optimize on demand."

- **Design for Scalability Early:** It's crucial to architect the system with **scalability in mind** from the outset. This means selecting architectural patterns (like Microservices or Event-Driven), technologies, and data storage solutions that are inherently capable of scaling. Changing fundamental architectural choices later to accommodate scalability is often extremely difficult and costly.

- **Optimize on Demand:** Avoid premature optimization. Fully optimizing for extreme scalability from day one for every component is wasteful. It adds unnecessary complexity, development time, and cost if the anticipated load never materializes. Instead, **identify bottlenecks** through monitoring and testing as the system grows, and then apply specific optimizations to those problematic areas. This "just-in-time" optimization saves resources and allows focus on immediate needs.

2. Can a modular monolith achieve the same team autonomy as microservices?

A modular monolith cannot achieve the same level of team autonomy as microservices, but it can significantly improve it compared to a traditional monolith.

- **Modular Monolith:** It allows teams to own and work independently on specific modules within the shared codebase. This provides autonomy over code within that module and reduces conflicts. However, teams still share a single deployment pipeline, potentially a single release schedule, and are constrained by the shared technology stack and overall build process. Coordination for major releases or infrastructure changes is still necessary.

- **Microservices:** Each microservice is an independent deployment unit, owned end-to-end by a single team. This provides **true operational and technological autonomy**. Teams can deploy their services on their own schedule, choose their own technology stack (within reason), and manage their own infrastructure, leading to maximum independence.

3. Are there systems where event-driven architecture causes more harm than good?

Yes, there are systems where Event-Driven Architecture (EDA) can cause more harm than good.

- **Systems Requiring Strong, Immediate Consistency:** EDAs inherently lead to **eventual consistency**. For systems where every transaction requires **absolute, real-time data consistency** across multiple components (e.g., core banking ledgers, critical inventory management where simultaneous updates must be globally consistent immediately), the complexity of managing eventual consistency, distributed transactions, and potential compensating actions can outweigh the benefits.

- **Simple, Small Applications with Limited Complexity:** For straightforward applications with minimal integration needs and low traffic, the **operational overhead and debugging complexity** of an EDA (managing message brokers, asynchronous flows, distributed tracing) can be disproportionate to the value it provides. A simpler monolithic or layered approach would be more efficient and easier to manage.

4. Should every modern application aim to use Microservices?

No, every modern application should not aim to use Microservices. While microservices offer significant benefits for certain types of applications, they introduce substantial complexity and are not a universal panacea.

- **Appropriate Use Cases:** Microservices are generally well-suited for **large, complex, evolving systems** that require independent scalability of components, technology diversity, and autonomous teams.

- **Drawbacks for Other Cases:** For **small, simple applications, startups with limited resources, or applications with stable requirements**, the overhead of managing a distributed system (increased complexity in development, testing, deployment, operations, and debugging) can far outweigh the benefits. A well-designed **monolith or modular monolith** is often a more pragmatic and efficient choice in these scenarios. The decision should be driven by genuine business needs and organizational capabilities, not just by trend.

5. How do you decide whether to prioritize maintainability or performance in an architecture?

Deciding whether to prioritize maintainability or performance depends entirely on the system's primary business drivers, user expectations, and long-term strategic goals.

Prioritize Performance When:

- **Real-time response is critical:** E.g., high-frequency trading, autonomous vehicle control, interactive gaming.
- **High throughput is essential:** E.g., advertising platforms, analytics systems processing vast data streams.

- **User experience is directly tied to speed:** E.g., e-commerce checkouts, search engines where even slight delays cause user abandonment.
- **Resource efficiency is a primary cost driver:** Minimizing server costs through highly optimized code.
- **Example:** For a financial trading system, a millisecond delay could mean millions of dollars lost. Performance would heavily outweigh some maintainability compromises.

Prioritize Maintainability When:

- **System longevity and evolution are key:** Most enterprise applications undergo continuous updates and feature additions over many years.
- **Cost of change needs to be low:** Frequent changes make an easily modifiable codebase more economical in the long run.
- **Team onboarding and productivity are important:** A well-structured, understandable codebase helps new developers become productive quickly.
- **System requirements are expected to change frequently:** Adaptability is crucial.
- **Example:** A complex internal ERP system will likely be maintained for decades. The ability to easily add modules or fix bugs without breaking existing functionality (high maintainability) would be far more valuable than shaving off a few milliseconds of response time for non-critical operations.

The Decision Process:

Often, it's not an "either/or" but a "how much of each." This requires:

- **Understanding Core Business Goals:** What's the system's primary purpose and critical success factors?
- **Defining Non-Functional Requirements:** Quantify desired performance targets (e.g., "response time < 200ms for 95% of requests") and maintainability goals (e.g., "new feature implementation < X days").
- **Risk Assessment:** What are the consequences of failing to meet each characteristic?
- **Trade-off Analysis:** Actively assess architectural options through the lens of these prioritized characteristics, understanding where compromises are acceptable.

END