WETWARE ENGINEERING: APPLYING SOFTWARE ENGINEERING PARADIGMS TO BIOLOGICAL SYSTEM CONSTRUCTION

A PREPRINT

123olp
tukuai.ai@gmail.com

December 29, 2025

ABSTRACT

Software engineering underwent a paradigm shift from monolithic programs to modular, composable systems over five decades—enabled by standardized interfaces, package managers, version control, and runtime orchestration. Biological engineering remains in an analogous "pre-modular" era: each system is constructed from scratch, results are difficult to reproduce, and no universal language exists for describing biological component composition.

We propose **Wetware Engineering**, a cross-disciplinary methodology that transfers software engineering's core abstractions—modularity, interface standardization, dependency management, and runtime orchestration—to biological system construction. Our contributions include: (1) the **Component-Interface-Runtime triad** as foundational abstraction for modular biological systems; (2) **Bio-Component Spec**, a standardized schema for describing biological modules; (3) **Bio-DSL**, a domain-specific language for declarative system composition; and (4) systematic analysis identifying both direct translations and fundamental differences requiring novel solutions.

Wetware Engineering represents a paradigm-level contribution: shifting biological system construction from "artisanal replication" to "engineered composition."

Keywords Software Engineering · Biological Systems · Modular Design · Domain-Specific Language · Systems Biology

1 Introduction: The Case for Paradigm Transfer

1.1 Software Engineering's Modular Revolution

The history of software engineering is fundamentally a history of rising abstraction levels. In the 1950s, programmers wrote machine code. Assembly language provided the first abstraction. Structured programming (1960s) abstracted control flow. Object-oriented programming (1980s) encapsulated data and behavior. Component-based development (1990s) enabled binary-level reuse. Service-oriented architecture (2000s) abstracted deployment. Microservices (2010s) achieved independent deployment and elastic scaling.

Each abstraction level did not merely add convenience—it fundamentally changed what was possible. Before package managers, sharing code meant copying files. Before containerization, "it works on my machine" was unsolvable. Before version control, collaboration meant emailing zip files.

Today, a developer can declare `import tensorflow` and instantly access millions of lines of tested, documented, version-controlled code—the accumulated result of decades of standardization.

1.2 Biological Engineering's "Pre-Modular" State

Biological engineering in 2025, despite extraordinary advances, remains analogous to software circa 1970:

Table 1: Software vs. Biological Engineering: Current State

Software Concept	Biology Status	The Gap
Standard Library	None	Each lab builds from scratch
Package Manager	None	Cannot declare dependencies
Version Control	None	"This batch differs from last"
API Documentation	None	"Ask the original author"
Unit Testing	None	"How long will it last?"

When a tissue engineer wants to combine a muscle actuator with a neural controller, they face challenges software engineers solved decades ago: no standard interfaces, no dependency declaration, no version compatibility, no composition language.

The fundamental problem is conceptual: biological systems are treated as **indivisible wholes** rather than **composable collections of modules**.

1.3 Why Paradigm Transfer, Not Just Tool Application

Existing "computational biology" means using computers to analyze data (bioinformatics), simulate processes (systems biology), or control experiments (automation). These apply software as a *tool* to biology.

We propose something different: **using software engineering's design philosophy to reconceptualize how biological systems are constructed**. This is paradigm transfer, not tool application.

1.4 Contributions

1. **Paradigm Definition**: Systematic transfer of software engineering paradigms to biological system construction
2. **Abstraction Framework**: Component-Interface-Runtime triad for modular biological systems
3. **Technical Specifications**: Bio-Component Spec v0.1 and Bio-DSL
4. **Mapping Analysis**: Direct, Analogous, and Novel mappings identified
5. **Difference Identification**: Fundamental challenges requiring innovation

2 Core Abstractions: The Component-Interface-Runtime Triad

2.1 Abstraction as the Essence of Engineering

Dijkstra observed: "The purpose of abstraction is not to be vague, but to create a new semantic level in which one can be absolutely precise." Software engineering's success stems from identifying correct abstraction boundaries: functions abstract instructions, objects abstract data and behavior, interfaces abstract implementations, containers abstract environments.

The central question for biological engineering: **What are the correct abstraction boundaries for living systems?**

We propose the same triad that revolutionized software: **Component**, **Interface**, and **Runtime**.

2.2 Component: The Unit of Biological Reuse

A **Bio-Component** is a biological unit that: can exist independently (with life support), can receive energy/nutrients, can respond to control signals, can produce functional outputs, and can report its state.

Table 2: Component Typology

Type	Software Analogy	Biological Examples
Actuator	Output driver	Muscle, gland
Sensor	Input driver	Photoreceptor, mechanoreceptor
Processor	CPU/logic	Ganglion, brain organoid
Storage	Memory/database	Adipose tissue
Connector	Network interface	Blood vessel, nerve
Metabolic	Power supply	Liver tissue

2.3 Interface: The Contract for Composition

The Gang of Four principle: "Program to an interface, not an implementation." Biological interfaces operate across four dimensions:

Power Interface: Energy and nutrient flow (perfusion connections, diffusion surfaces)

Signal Interface: Information exchange (electrical, chemical, mechanical, optical)

Isolation Interface: Protection barriers (immune, toxicity, electrical isolation)

Mechanical Interface: Force transmission (structural attachments, movement coupling)

The USB analogy: before USB, every device had proprietary connectors. After USB, one connector fits all. We envision Bio-Interfaces achieving similar transformation.

2.4 Runtime: The Orchestration Layer

A Bio-Runtime handles: nutrient allocation (memory), activity timing (CPU scheduling), signal routing (network I/O), viability monitoring (health checks), regeneration triggering (auto-restart), biomarker recording (logging), and fault isolation (containing necrosis).

The perfusion system—delivering nutrients and oxygen while removing waste—is the biological equivalent of power and network infrastructure.

2.5 The Triad in Action

A simple bio-robotic system (muscle responding to force) in Bio-DSL:

```
CONNECT sensor.output TO controller.input
CONNECT controller.output TO muscle.stimulation
RUNTIME { perfusion: standard_mammalian, control: closed_loop }
```

The same description works with different implementations (human/mouse/synthetic muscle, piezoelectric/biological sensor) as long as interfaces are honored.

3 Systematic Mapping: Software to Biological Engineering

3.1 Mapping Framework

We categorize mappings as:

- **Direct**: Transfers with minimal adaptation (versioning, dependencies, docs)
- **Analogous**: Core idea transfers but requires adaptation (testing, error handling)
- **Novel**: No software equivalent; requires new solutions (immune, degradation)

3.2 Direct Mappings

Semantic Versioning: MAJOR.MINOR.PATCH transfers directly. Example: `muscle-actuator@2.3.1` where MAJOR=interface-breaking, MINOR=backward-compatible additions, PATCH=fixes.

Dependency Declaration: Package manifests work identically:

```
dependencies:
  perfusion-medium: "DMEM@^1.0"
  oxygen-supply: ">=15%"
```

Documentation and Licensing: README files, API docs, and license frameworks transfer directly.

3.3 Analogous Mappings

Testing Validation: Software tests are deterministic and fast; biological tests are statistical and slow. Adaptation: define acceptance criteria as statistical thresholds.

Error Handling Failure Mode Management: Software has exceptions/errors/warnings; biology has recoverable degradation (fatigue), irreversible damage (necrosis), and systemic risk (inflammation).

Logging Biomarker Recording: Timestamps, measurements, environmental conditions, anomaly indicators.

3.4 Novel Challenges

Immune Compatibility: Software components don't "reject" each other. Required: compatibility scoring, isolation barriers.

Signal Crosstalk: Software processes are isolated; biological components share chemical environments.

Metabolic Coupling: Software components consume resources independently; biological components share metabolites and produce waste affecting neighbors.

Living Degradation: Software doesn't age; biological components inherently degrade.

Ethical Constraints: Software has no ethical status; biological components raise consent, capability, and disposal considerations.

4 Bio-Component Specification

4.1 Design Principles

Bio-Component Spec draws from software engineering principles:

SOLID Applied: Single Responsibility (one biological function), Open/Closed (enhance without changing interfaces), Liskov Substitution (compatible components interchangeable), Interface Segregation (fine-grained definitions), Dependency Inversion (depend on specs, not implementations).

Convention over Configuration: Sensible defaults minimize required configuration.

Schema-First Design: Define schema before implementations, like OpenAPI for REST APIs.

4.2 Specification Structure

The complete schema (Appendix C) includes:

- **Identification**: id, name, version, license, authors
- **Classification**: type (actuator/sensor/processor/...), domain, tags
- **Source**: organism, tissue type, cell types, biosafety level
- **Interface**: input/output port definitions with parameters
- **Requirements**: physical (temperature), chemical (pH, oxygen), biological
- **Performance**: functional metrics, reliability, resource consumption
- **Failure Modes**: detection, type (recoverable/irreversible), mitigation
- **Testing**: unit tests, integration tests
- **Dependencies**: components, adapters, protocols

4.3 Versioning Strategy

Semantic versioning with biological interpretations:

- **MAJOR**: Interface-breaking (port type changes, new required parameters)
- **MINOR**: Backward-compatible (new optional ports, performance improvements)
- **PATCH**: Fixes (protocol optimizations, documentation)

Extended format for biological specificity: `2.3.1+batch20251228.donor42.wildtype`

A complete muscle actuator specification is provided in Appendix B.

5 Bio-DSL: Language Design

5.1 Why a Domain-Specific Language?

DSLs trade generality for expressiveness. Benefits for biological systems:

- **Expressiveness**: Describe complex assemblies concisely
- **Readability**: Biologists can understand without programming background
- **Validation**: Catch interface mismatches before physical assembly
- **Abstraction**: Focus on what, not how

5.2 Design Goals

1. **Declarative**: Describe *what* the system is, not *how* to build it
2. **Readable**: Biologists should understand the intent
3. **Verifiable**: Static analysis catches errors before assembly
4. **Executable**: Generates runtime configurations
5. **Composable**: Systems can be nested and reused

5.3 Language Constructs

Component Declaration:

```
COMPONENT flexor FROM "muscle-actuator@^2.3"
COMPONENT sensor FROM "piezo-sensor@~1.1" AS force_sensor
```

Connection Declaration:

```
CONNECT sensor.output TO controller.input
CONNECT controller.output TO muscle.stim VIA adapter
```

Runtime Configuration:

```
RUNTIME {
  perfusion: { temp: 37C, pH: 7.4 },
  control: { mode: "closed_loop" }
}
```

Behavioral Logic:

```
ON STARTUP DO { SET perfusion.flow = 0.5; WAIT 300s }
WHEN fatigue > 0.3 THEN { REDUCE freq BY 20% }
```

Test Declaration:

```
TEST response {
  GIVEN muscle.state == "ready"
  WHEN STIMULATE AT 10Hz FOR 1s
  THEN EXPECT force IN [5,15] mN
}
```

5.4 Relationship to Existing Languages

Bio-DSL complements lower-level languages:

- **SBOL**: Genetic level (DNA sequences)—component internals
- **SBML**: Molecular level (biochemical networks)—component dynamics
- **CellML**: Cellular level (mathematical models)—behavior models
- **Bio-DSL**: Organ/System level—component composition

A complete dual-muscle antagonist system example is provided in Appendix A.

6 Fundamental Differences and Open Challenges

While paradigm transfer is powerful, fundamental differences require novel solutions:

6.1 Determinism vs. Stochasticity

Software: `add(2,3)` always returns 5. Biology: identical stimulation produces variable force, occasionally no response.

Implications: Probabilistic interface contracts, statistical testing, redundant components, graceful degradation.

6.2 Discrete vs. Continuous

Software state transitions are instantaneous (true/false). Biological transitions are gradual (relaxed contracting contracted).

Implications: Tolerance ranges for parameters, threshold-based state definitions, range-based timing specifications.

6.3 Isolation vs. Coupling

Software processes are isolated by OS memory protection. Biological components share culture medium; one component's waste affects all others.

Implications: Explicit coupling declarations, isolation adapters, system-level resource budgeting.

6.4 The Immune System

Software components don't "reject" each other. Biological components from different genetic backgrounds trigger immune responses.

Required Innovations: Immune compatibility scoring, isolation barrier specifications, compatibility checking in Bio-DSL.

6.5 Living Degradation

Software doesn't age. Biological components inherently degrade: cells senesce, proteins denature, structures weaken.

Required Innovations: Degradation modeling (Weibull distribution), maintenance protocols, hot-swap replacement strategies.

6.6 Ethical Constraints

Software has no ethical status. Biological components raise considerations: source ethics (consent), capability ethics (consciousness), use ethics (applications), disposal ethics.

Required Innovations: Ethical metadata in specifications, capability limits enforced by compiler.

6.7 Summary: Innovation Agenda

Table 3: Challenges Requiring Innovation

Challenge	Software	Required Innovation
Stochasticity	Deterministic	Probabilistic contracts
Continuous	Discrete	Tolerance ranges
Coupling	Isolated	Resource budgeting
Immune	None	Compatibility scoring
Degradation	None	Maintenance protocols
Ethics	Licensing	Capability limits

These challenges define the research agenda, not invalidate the paradigm transfer.

7 Related Work and Positioning

7.1 Synthetic Biology

BioBricks/iGEM pioneered standardization at the genetic level. SBOL provides data formats for genetic designs.

Relationship: BioBricks/SBOL operate at genetic level; Wetware Engineering at organ/system level. Complementary: BioBricks defines genetic content *within* Bio-Components.

7.2 Organ-on-Chip and Organoids

Organ-on-chip devices culture cells in microfluidic environments. Organoids are self-organizing 3D tissue cultures.

Relationship: These provide physical implementations but lack standardized interfaces. Wetware Engineering provides the abstraction framework.

7.3 Systems Biology Modeling

SBML/CellML describe how components behave internally (simulation).

Relationship: Bio-DSL describes how components connect externally (composition). Different purposes, can be used together.

7.4 Biohybrid Robotics

Research groups have created muscle-powered swimmers, biohybrid grippers. These prove feasibility.

Relationship: Current work is bespoke. Wetware Engineering provides systematic, reproducible methodology.

7.5 What We Are NOT Claiming

1. We have not built working systems (framework proposal)
2. Biology is not "just like" software (Section 6 details differences)
3. We don't replace existing approaches (we complement them)
4. This is not immediately practical (roadmap spans decades)

What we claim: Software engineering's conceptual framework provides valuable abstractions for biological system construction.

8 Conclusion and Future Directions

8.1 Summary of Contributions

This paper proposed **Wetware Engineering**, transferring software engineering paradigms to biological system construction:

- **Conceptual Framework**: Component-Interface-Runtime triad
- **Technical Specifications**: Bio-Component Spec v0.1, Bio-DSL
- **Systematic Analysis**: Direct, Analogous, Novel mappings
- **Honest Assessment**: Fundamental differences requiring innovation

8.2 The Path Forward

Phase 1 (1-3 years): Refine specifications, proof-of-concept tooling, document existing systems, reference implementations.

Phase 2 (3-7 years): Physical implementations, reproducibility validation, interface adapters, component registry.

Phase 3 (7-15 years): Community governance, commercial adoption, educational curriculum, international standardization.

8.3 Call to Action

Biologists: Describe work using Bio-Component Spec; identify interface requirements; share machine-readable protocols.

Software Engineers: Contribute tooling; apply design patterns; develop testing frameworks.

Standards Bodies: Engage early; coordinate with SBOL/SBML.

Funding Agencies: Support cross-disciplinary methodology; fund infrastructure.

8.4 Closing Thoughts

Software engineering transformed from craft to discipline over five decades through conceptual breakthroughs, standardization, tooling, and community building.

Biological engineering stands at a similar inflection point. The question is not whether modularization will come—complexity demands it—but how quickly and how well.

The tools of software engineering—abstraction, modularity, standardization, composition—are not specific to silicon. They are **general principles of managing complexity**. Biology is complex. These principles can help.

The future of biological engineering is modular. Will we design that future deliberately, or stumble into it accidentally?

We choose to design.

Data Availability

All specifications and examples: `https://github.com/tukuaiai/wetware-engineering`

References

[1] Parnas, D.L. (1972). On the criteria to be used in decomposing systems into modules. *Commun. ACM*, 15(12), 1053-1058.

[2] Dijkstra, E.W. (1968). Go to statement considered harmful. *Commun. ACM*, 11(3), 147-148.

[3] Gamma, E., Helm, R., Johnson, R., Vlissides, J. (1994). *Design Patterns*. Addison-Wesley.

[4] Fowler, M. (2010). *Domain-Specific Languages*. Addison-Wesley.

[5] Endy, D. (2005). Foundations for engineering biology. *Nature*, 438, 449-453.

[6] Galdzicki, M. et al. (2014). The Synthetic Biology Open Language (SBOL). *Nature Biotechnol.*, 32(6), 545-550.

[7] Lancaster, M.A., Knoblich, J.A. (2014). Organogenesis in a dish. *Science*, 345, 1247125.

[8] Bhatia, S.N., Ingber, D.E. (2014). Microfluidic organs-on-chips. *Nature Biotechnol.*, 32(8), 760-772.

[9] Hucka, M. et al. (2003). The systems biology markup language (SBML). *Bioinformatics*, 19(4), 524-531.

[10] Lloyd, C.M., Halstead, M.D., Nielsen, P.F. (2004). CellML: its future, present and past. *Prog. Biophys. Mol. Biol.*, 85, 433-450.

[11] Raman, R., Bashir, R. (2017). Biomimicry, biofabrication, and biohybrid systems. *Adv. Healthcare Mater.*, 6(20), 1700496.

[12] Ricotti, L. et al. (2017). Biohybrid actuators for robotics. *Science Robotics*, 2(12), eaaq0495.

[13] Park, S.J. et al. (2016). Phototactic guidance of a tissue-engineered soft-robotic ray. *Science*, 353, 158-162.

[14] Yuste, R. et al. (2017). Four ethical priorities for neurotechnologies and AI. *Nature*, 551, 159-163.

[15] Hyun, I., Scharf-Deering, J.C., Lunshof, J.E. (2020). Ethical issues related to brain organoid research. *Brain Res.*, 1732, 146653.

A Complete Bio-DSL Example: Dual-Muscle Antagonist System

```
// ============================================
// Bio-Mechanical Arm Unit v0.1
// Dual-muscle antagonist with closed-loop control
// ============================================

// === Component Declarations ===
COMPONENT flexor FROM "muscle-actuator-human-skeletal@^2.3" {
  role: "agonist",
  force_range: [0, 50] mN
}

COMPONENT extensor FROM "muscle-actuator-human-skeletal@^2.3" {
  role: "antagonist",
  force_range: [0, 50] mN
}

COMPONENT sensor FROM "piezo-force-sensor@~1.1" {
  range: [0, 100] mN,
  sampling_rate: 100 Hz
}

COMPONENT controller FROM "neural-organoid-spinal@>=0.8" {
  input_channels: 2,
  output_channels: 2
}

// === Adapter Declarations ===
ADAPTER perfusion FROM "microfluidic-4ch@1.0" {
  medium: "DMEM + 10% FBS",
  flow_rate: 0.5 mL/min PER channel
}

ADAPTER stim_converter FROM "opto-electrical@2.0" {
  wavelength: 470 nm
}

// === Connection Topology ===
CONNECT controller.output_1 TO flexor.stimulation
  VIA stim_converter
  WITH { frequency: [1, 50] Hz, voltage: [0, 3] V }

CONNECT controller.output_2 TO extensor.stimulation
  VIA stim_converter
  WITH { frequency: [1, 50] Hz, voltage: [0, 3] V }

CONNECT sensor.output TO controller.feedback_input
  WITH { gain: 1.5 }

CONNECT perfusion.ch1 TO flexor.perfusion_input
CONNECT perfusion.ch2 TO extensor.perfusion_input
CONNECT perfusion.ch3 TO controller.perfusion_input
```

```
CONNECT perfusion.ch4 TO sensor.perfusion_input

// === Runtime Configuration ===
RUNTIME {
  perfusion: {
    temperature: 37 C,
    pH: 7.4,
    oxygenation: true,
    waste_removal: "continuous"
  },

  control: {
    mode: "closed_loop",
    target: "position",
    pid: { Kp: 0.8, Ki: 0.2, Kd: 0.1 }
  },

  monitoring: {
    interval: 10 s,
    metrics: [
      "flexor.force", "extensor.force",
      "sensor.reading", "controller.activity",
      "*.viability"
    ],
    alerts: {
      "viability < 80%": "WARNING",
      "force > 90 mN": "CRITICAL"
    }
  },

  safety: {
    max_force: 100 mN,
    emergency: {
      trigger: "viability < 50% OR force > 120 mN",
      action: "STOP_ALL; PERFUSION_ONLY"
    }
  }
}

// === Behavioral Logic ===
ON STARTUP DO {
  SET perfusion.flow_rate = 0.5 mL/min
  WAIT 300 s  // Equilibration period
  RUN calibration_sequence()
  SET controller.mode = "active"
  LOG "System initialized"
}

WHEN target_position CHANGES THEN {
  error = target_position - sensor.reading
  controller.compute(error)
}

WHEN flexor.fatigue_index > 0.3 THEN {
  LOG "Flexor fatigue detected"
  REDUCE flexor.stim_frequency BY 20%
  INCREASE extensor.stim_frequency BY 10%
}

EVERY 1 hour DO {
  RUN viability_check()
  RECORD performance_snapshot()
}

// === Test Suite ===
```

```
TEST unit_response {
  description: "Single muscle contraction test"
  GIVEN flexor.state == "ready"
  WHEN STIMULATE flexor AT 10 Hz, 2 V FOR 1 s
  THEN EXPECT flexor.force IN [5, 15] mN WITHIN 200 ms
}

TEST antagonist_balance {
  description: "Antagonist coordination test"
  GIVEN system.mode == "active"
  WHEN ACTIVATE flexor AND extensor SIMULTANEOUSLY
  THEN EXPECT |flexor.force - extensor.force| < 5 mN
}

TEST tracking_accuracy {
  description: "Closed-loop tracking test"
  GIVEN system.mode == "closed_loop"
  WHEN SET target = sine_wave(0.5 Hz, +/-20 mN) FOR 60 s
  THEN EXPECT tracking_error < 3 mN RMS
}

// === Expected Performance ===
EXPECTED {
  position_accuracy: +/-2 mN,
  bandwidth: [0, 2] Hz,
  lifetime: [7, 14] days,
  power_consumption: [50, 100] mW
}
```

B Complete Muscle Actuator Specification

```
bio-component: "1.0"

info:
  id: "muscle-actuator-human-skeletal"
  name: "Human Skeletal Muscle Actuator"
  version: "2.3.1"
  description: "Contractile muscle tissue for force generation"
  license: "CC-BY-SA-4.0"
  authors: ["Wetware Engineering Project"]
  repository: "https://github.com/wetware/muscle-actuator"

classification:
  type: "actuator"
  domain: "musculoskeletal"
  tags: ["muscle", "contractile", "force-generation", "human"]

source:
  organism: "Homo sapiens"
  tissue_type: "skeletal muscle"
  cell_types: ["myocyte", "satellite cell"]
  culture_protocol: "https://protocols.io/wetware/muscle-culture-v2"
  biosafety_level: "BSL-1"

interface:
  inputs:
    - id: "electrical_stimulation"
      type: "electrical"
      required: true
      parameters:
        voltage: { range: [0, 5], unit: "V" }
        frequency: { range: [1, 100], unit: "Hz" }
        pulse_width: { range: [0.1, 10], unit: "ms" }
```

```
      response_time: 50  # ms

    - id: "perfusion_input"
      type: "perfusion"
      required: true
      parameters:
        flow_rate: { range: [0.1, 2.0], unit: "mL/min" }

  outputs:
    - id: "force_output"
      type: "mechanical"
      parameters:
        force: { range: [0, 50], unit: "mN" }
        displacement: { range: [0, 5], unit: "mm" }
      latency: 150  # ms
      monitoring:
        metrics: ["force", "displacement", "velocity"]
        sampling_rate: 100  # Hz

requirements:
  physical:
    temperature: { optimal: 37, range: [35, 39], unit: "C" }
  chemical:
    pH: { optimal: 7.4, range: [7.2, 7.6] }
    oxygen: { range: [15, 25], unit: "%" }
    glucose: { range: [5, 25], unit: "mM" }

performance:
  functional:
    max_force: { value: 50, unit: "mN" }
    response_time: { typical: 150, max: 300, unit: "ms" }
    contraction_velocity: { max: 10, unit: "mm/s" }
  reliability:
    lifetime: { mean: 14, std: 3, unit: "days" }
    failure_rate: { value: 0.01, unit: "per_hour" }
  resources:
    oxygen_consumption: { value: 5, unit: "umol/hour" }
    glucose_consumption: { value: 2.5, unit: "umol/hour" }

failure_modes:
  - id: "fatigue"
    type: "recoverable"
    probability: 0.3
    detection: "force_output < 80% baseline"
    impact: "reduced_performance"
    mitigation: "reduce stimulation, allow recovery"

  - id: "necrosis"
    type: "irreversible"
    probability: 0.05
    detection: "viability < 50%"
    impact: "component_loss"
    mitigation: "replace component"

testing:
  unit_tests:
    - id: "contraction_response"
      description: "Verify force output on stimulation"
      protocol: "stimulate 10Hz 2V for 1s, measure force"
      acceptance: "force in [5, 15] mN within 200ms"

    - id: "viability_check"
      description: "Verify cell survival"
      protocol: "live/dead staining"
      acceptance: "viability > 90%"
```

```yaml
dependencies:
  adapters:
    - id: "perfusion-adapter"
      version: "^1.0.0"
  protocols:
    - name: "standard-mammalian-culture"
      version: "2.1"
```

C Bio-Component Spec JSON Schema

```json
{
  "$schema": "https://wetware-engineering.org/schema/bio-component/1.0",
  "type": "object",
  "required": ["bio-component", "info", "classification", "source", "interface"],
  "properties": {
    "bio-component": {
      "type": "string",
      "pattern": "^\\d+\\.\\d+$"
    },
    "info": {
      "type": "object",
      "required": ["id", "name", "version"],
      "properties": {
        "id": {"type": "string", "pattern": "^[a-z0-9-]+$"},
        "name": {"type": "string"},
        "version": {"type": "string", "pattern": "^\\d+\\.\\d+\\.\\d+"},
        "description": {"type": "string"},
        "license": {"type": "string"},
        "authors": {"type": "array", "items": {"type": "string"}},
        "repository": {"type": "string", "format": "uri"}
      }
    },
    "classification": {
      "type": "object",
      "properties": {
        "type": {
          "type": "string",
          "enum": ["actuator", "sensor", "processor",
                   "metabolic", "structural", "connector"]
        },
        "domain": {"type": "string"},
        "tags": {"type": "array", "items": {"type": "string"}}
      }
    },
    "source": {
      "type": "object",
      "properties": {
        "organism": {"type": "string"},
        "tissue_type": {"type": "string"},
        "cell_types": {"type": "array", "items": {"type": "string"}},
        "cell_line": {"type": "string"},
        "genetic_modifications": {"type": "array"},
        "culture_protocol": {"type": "string", "format": "uri"},
        "biosafety_level": {"enum": ["BSL-1", "BSL-2", "BSL-3", "BSL-4"]}
      }
    },
    "interface": {
      "type": "object",
      "properties": {
        "inputs": {"type": "array", "items": {"$ref": "#/defs/Port"}},
        "outputs": {"type": "array", "items": {"$ref": "#/defs/Port"}}
      }
```

```
    },
    "requirements": {
      "type": "object",
      "properties": {
        "physical": {"$ref": "#/defs/PhysicalReq"},
        "chemical": {"$ref": "#/defs/ChemicalReq"},
        "biological": {"$ref": "#/defs/BiologicalReq"}
      }
    },
    "performance": {
      "type": "object",
      "properties": {
        "functional": {"type": "object"},
        "reliability": {"type": "object"},
        "resources": {"type": "object"}
      }
    },
    "failure_modes": {
      "type": "array",
      "items": {"$ref": "#/defs/FailureMode"}
    },
    "testing": {
      "type": "object",
      "properties": {
        "unit_tests": {"type": "array"},
        "integration_tests": {"type": "array"}
      }
    },
    "dependencies": {
      "type": "object",
      "properties": {
        "components": {"type": "array"},
        "adapters": {"type": "array"},
        "protocols": {"type": "array"}
      }
    }
  }
}
```