Microservices

The Good, the Bad, and the Ugly

James Foster — ESUG 2025 — Gdansk, Poland

Motivation

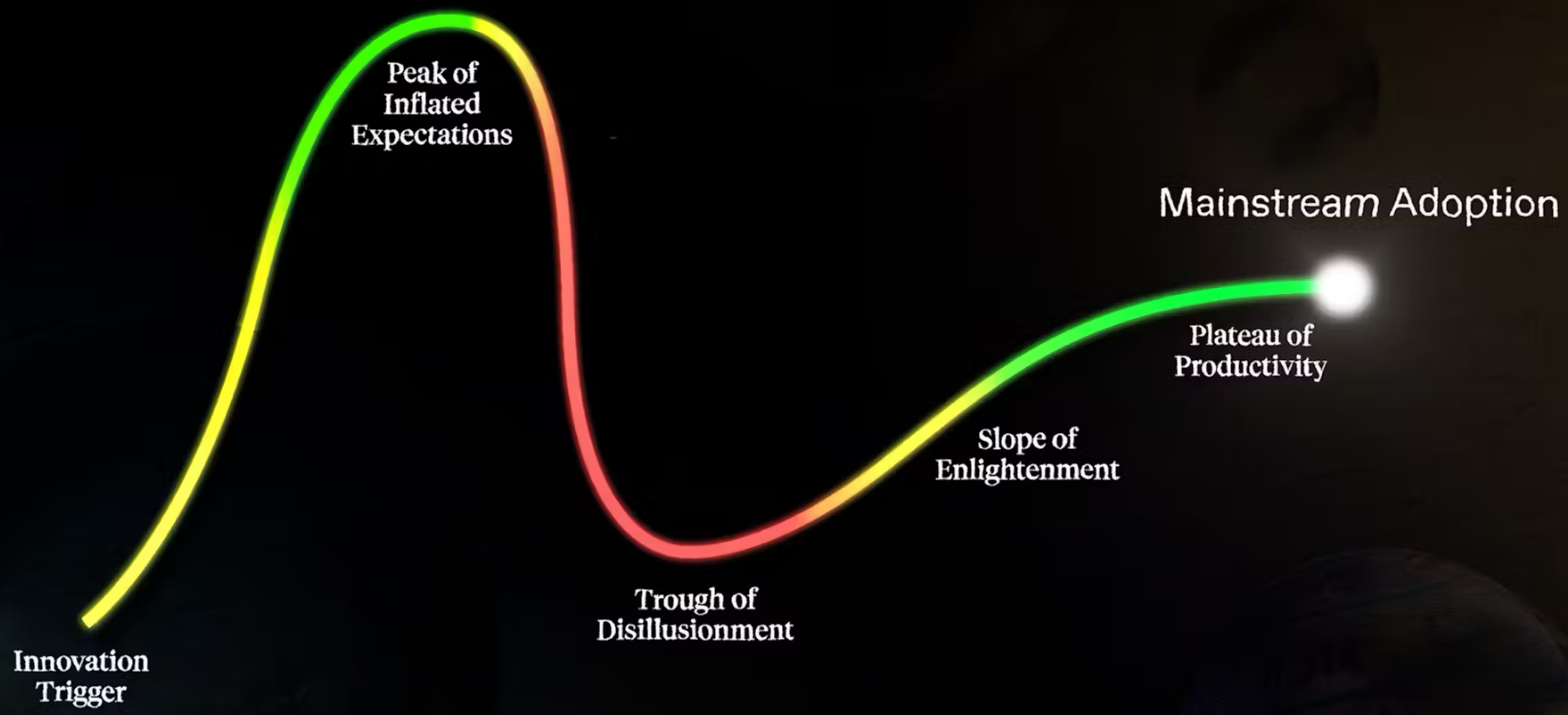
- GemTalk customer expressing interest in breaking monolith into microservices
- ESUG 2025 Call for Presentations (https://esug.org/2025-Conference/agenda.html):
  - The list of topics for the normal talks and tutorials includes, but is not limited to the following:
    - *Micro Services*, Container, Cloud, Big Data

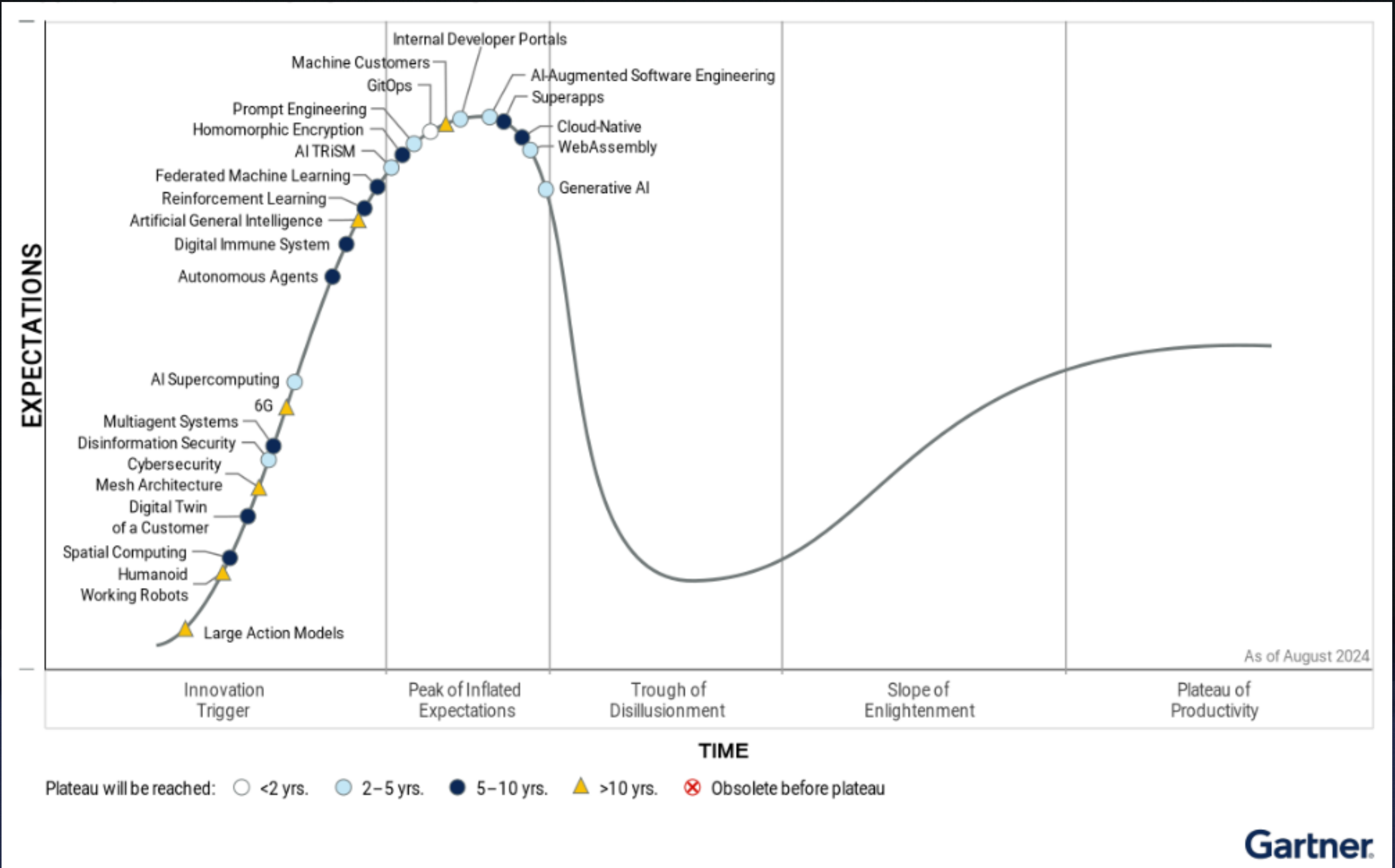
Agenda

- The Gartner Hype Cycle
- Modularity
- Monolith
- Serverless Computing
- Microservices
- GemStone/S 64 Bit

The Gartner Hype Cycle

The Gartner Hype Cycle

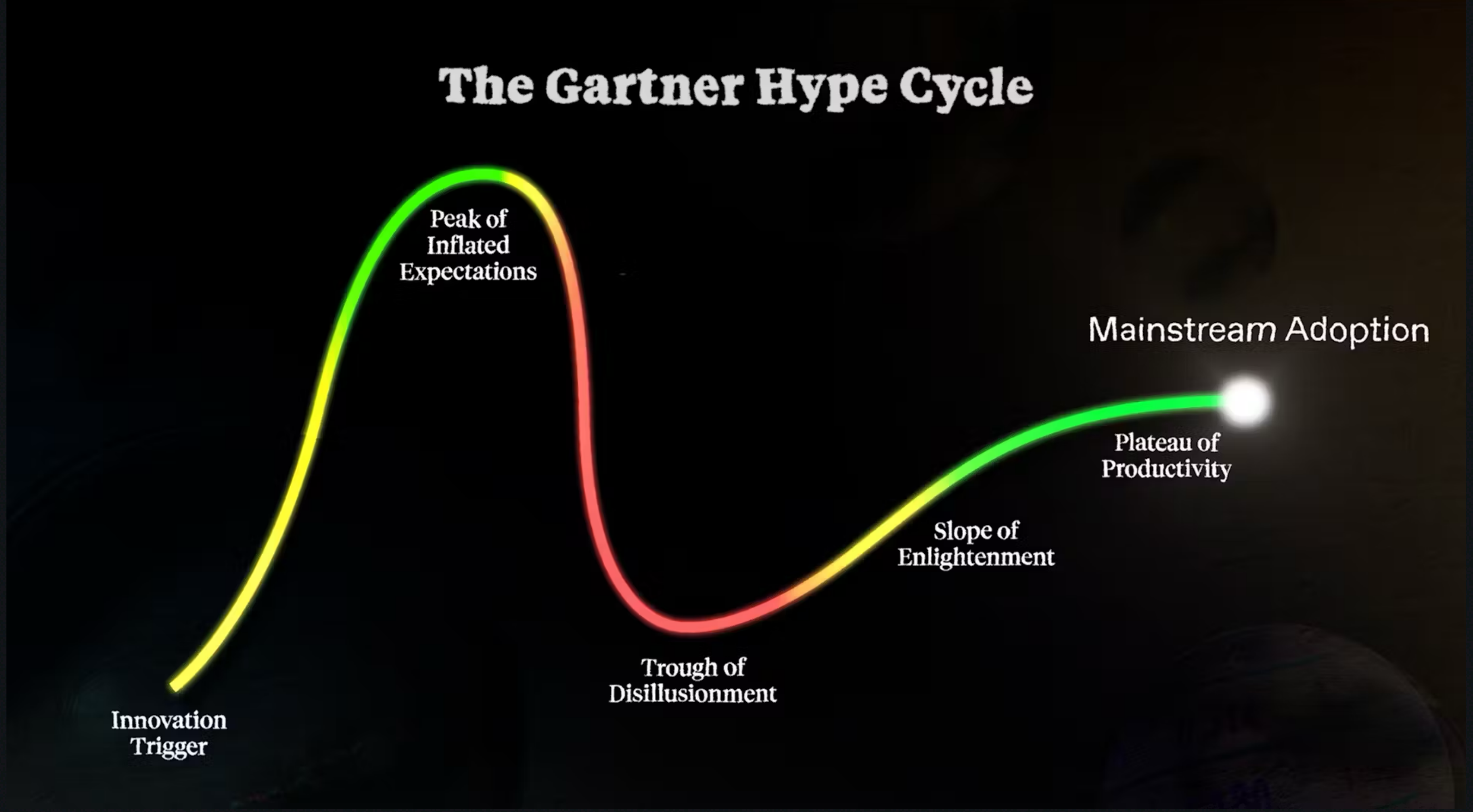




EXPECTATIONS
Internal Developer Portals
Machine Customers
GitOps
AI-Augmented Software Engineering
Superapps
Prompt Engineering
Homomorphic Encryption
Cloud-Native
WebAssembly
AI TRiSM
Federated Machine Learning
Generative AI
Reinforcement Learning
Artificial General Intelligence
Digital Immune System
Autonomous Agents
AI Supercomputing
6G
Multiagent Systems
Disinformation Security
Cybersecurity
Mesh Architecture
Digital Twin
of a Customer
Spatial Computing
Humanoid
Working Robots
Large Action Models
As of August 2024
Innovation
Trigger
Peak of Inflated
Expectations
Trough of
Disillusionment
Slope of
Enlightenment
Plateau of
Productivity
TIME
Plateau will be reached: <2 yrs. 2–5 yrs. 5–10 yrs. >10 yrs. Obsolete before plateau
Gartner

The Gartner Hype Cycle

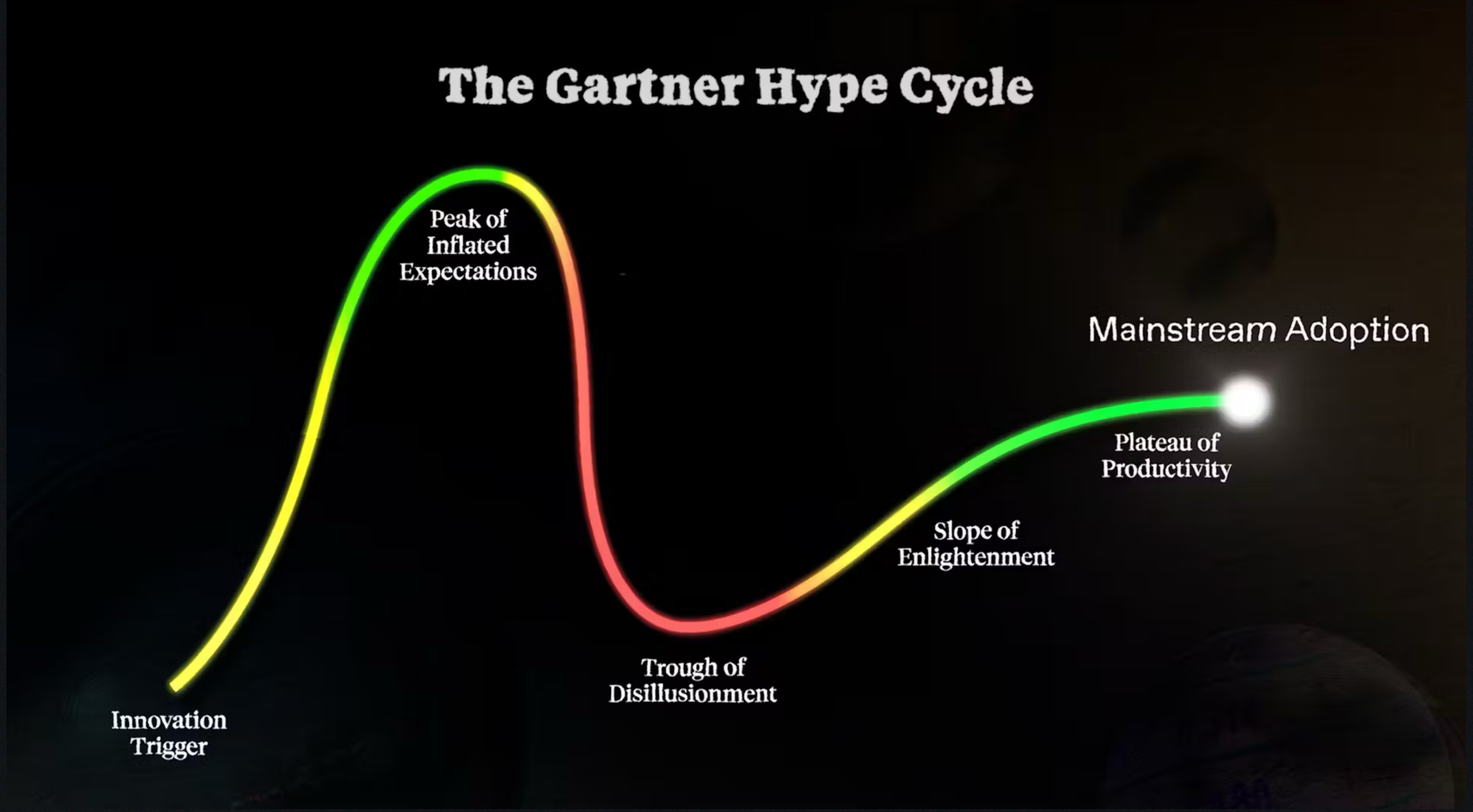
Microservices



- Innovation Trigger (2014)
  - Microservices began gaining attention as a new architectural style, especially with the rise of DevOps and cloud-native development.
- Peak of Inflated Expectations (2015-2016)
  - Hype surged as companies like Netflix and Amazon showcased success. Many organizations rushed to adopt without fully understanding the complexity.

The Gartner Hype Cycle

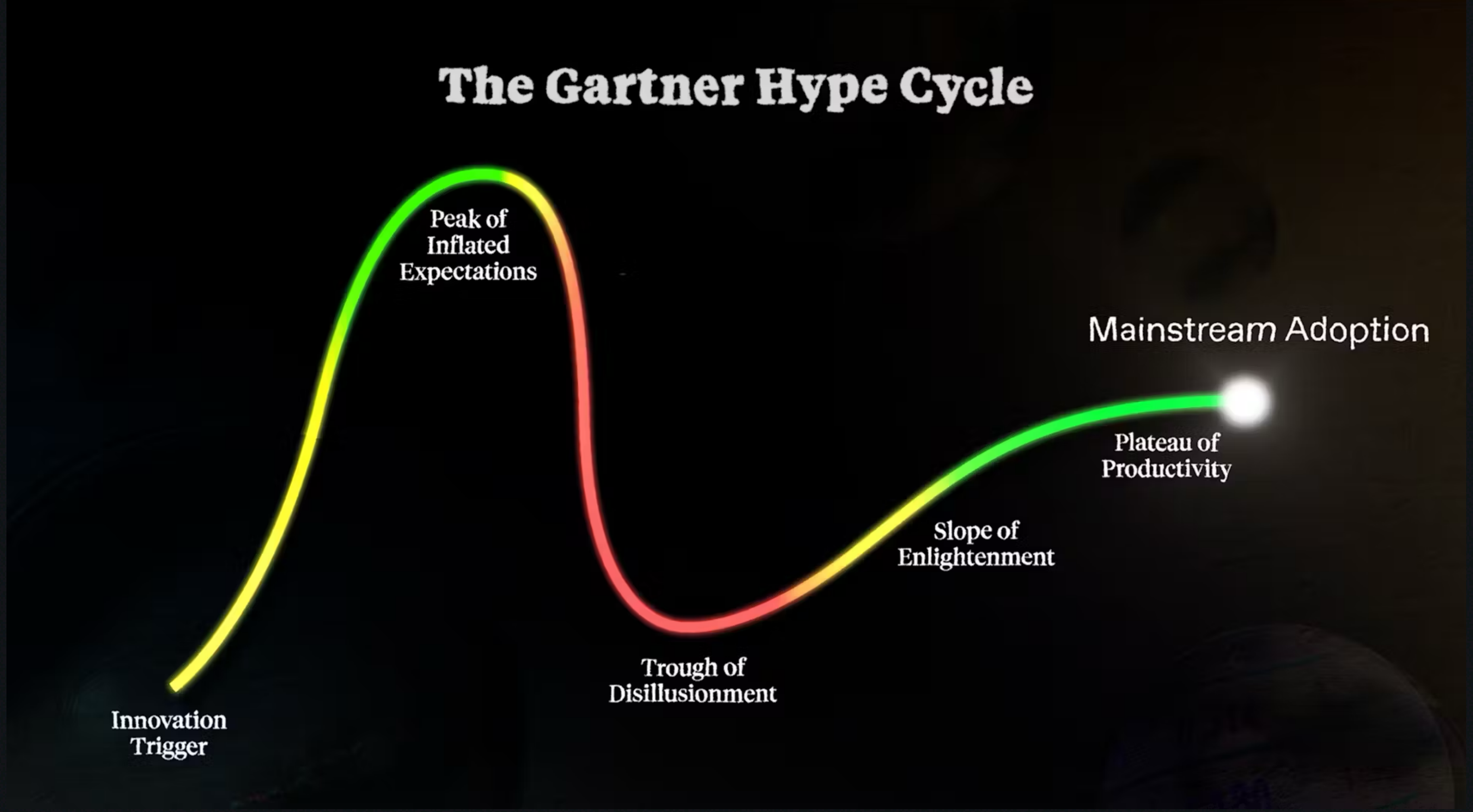
Microservices



- Trough of Disillusionment (2017-2018)
  - Challenges such as service sprawl, monitoring, and deployment complexity became apparent. Some early adopters struggled with implementation.
- Slope of Enlightenment (2019-2021)
  - Best practices, tooling (e.g., Kubernetes, service meshes), and patterns matured. Organizations began to understand when and how to use microservices effectively.

The Gartner Hype Cycle

Microservices



- Plateau of Productivity (2022-2025)
  - Microservices are now a mainstream architectural choice, especially in large-scale and cloud-native systems. Adoption is widespread, but often combined with modular monoliths or hybrid approaches.

Modularity

Modularity

Definition

- Modularity in software refers to the design principle of breaking down a software system into separate, interchangeable, and self-contained components or "modules."
- Each module encapsulates a specific functionality and interacts with other modules through well-defined interfaces.
- This approach enhances maintainability, reusability, scalability, and collaborative development.

Modularity

1950s–1960s: Early Concepts

- Assembly and early high-level languages (like FORTRAN) had monolithic structures.
- The idea of structured programming emerged, emphasizing control structures and subroutines (e.g., in ALGOL).

Modularity

1970s: Formalization of Modularity

- David Parnas (1972) published the seminal paper *On the Criteria To Be Used in Decomposing Systems into Modules*, advocating for information hiding as a key to modular design.
- Modula and Modula-2 (by Niklaus Wirth) were early languages explicitly supporting modular programming.

Modularity

1980s–1990s: Object-Oriented Programming (OOP)

- OOP languages like Smalltalk, C++, and later Java introduced modularity through classes and encapsulation.
- Component-based software engineering (CBSE) gained traction, promoting reusable software components.

Modularity

2000s–2010s: Modular Architectures

- Service-Oriented Architecture (SOA) and microservices became popular, emphasizing modular services communicating over networks.
- Package managers (like npm, pip, Maven) facilitated modular development and distribution.

Modularity

2020s–Present: Modular Ecosystems

- Languages like Rust, Go, and modern JavaScript emphasize modularity through packages, crates, and modules.
- Modularity in cloud-native applications:
    - Modular monoliths
    - Serverless computing
    - Microservices

Modularity

Benefits

- Easier debugging and testing
- Parallel development by teams
- Code reuse across projects
- Simplified maintenance and updates

Monolith

Monolith

What is a Monolith?

- A single runtime executable that contains all relevant application code.
- Changes to any module require deployment of entire application.

Monolith

Advantages - 1

- Simplicity of Development
  - Easier to set up and understand, especially for small teams or projects.
  - No need to manage inter-service communication or distributed systems complexity.
- Easier Testing
  - End-to-end testing is more straightforward since everything runs in a single process.

Monolith

Advantages - 2

- Performance
  - In-process calls are faster than network calls between microservices.
  - No serialization/deserialization overhead.
- Simplified Deployment
  - One deployment pipeline and runtime environment.
  - No need for service discovery, orchestration, or containerization.

Monolith

Advantages - 3

- Centralized Management
  - Easier to manage logging, monitoring, and debugging in a single codebase and process.

Monolith

Disadvantages - 1

- Scalability Limitations
    - You can only scale the entire application, not individual components.
- Tight Coupling
    - Changes in one part of the system can affect others, making it harder to maintain.
- Slower Development at Scale
    - As the codebase grows, onboarding new developers and managing dependencies becomes harder.

Monolith

Disadvantages - 2

- Deployment Risks
  - A bug in one module can bring down the entire application.
  - Frequent deployments are riskier and harder to coordinate.
- Technology Lock-In
  - Harder to use different technologies or languages for different parts of the system.

Monolith

Modular Monolith

- Good architecture will have discrete modules/libraries with function calls between them.
- Module vs. Library
  - Module is internally developed
  - Library is externally developed

Monolith

Advantages of a Modular Monolith

- Encourages clean architecture: Modules enforce separation of concerns.
- Easier to refactor: You can extract modules into microservices later if needed.
- Simplifies deployment: Still a single deployable unit.
- Improves team collaboration: Teams can work on different modules with minimal interference.

Monolith

Disadvantages of a Modular Monolith

- Internal interdependencies: Changes to a module may require changes to others.
- Still a single point of failure: A bug in one module can affect the whole system.
- Scaling is coarse-grained: You can't scale modules independently.
- Requires discipline: Developers must respect module boundaries to avoid tight coupling.

Serverless Computing

Serverless Computing

Description

- Serverless computing is a cloud computing model where developers build and run applications without having to manage the underlying infrastructure.
- Despite the name, it doesn't mean there are no servers—it means that server management is abstracted away and handled entirely by the cloud provider.

Serverless Computing

Key Characteristics - 1

- No Server Management
  - Developers don’t provision, scale, or maintain servers. The cloud provider handles all of that automatically.
- Event-Driven Execution
  - Code runs in response to events (e.g., HTTP requests, file uploads, database changes).
- Automatic Scaling
  - The platform automatically scales the application up or down based on demand.

Serverless Computing

Key Characteristics - 2

- Pay-as-You-Go
  - You're billed only for the compute time your code actually uses—no charges for idle time.
- Short-Lived Functions
  - Often implemented using Functions-as-a-Service (FaaS), like AWS Lambda, Azure Functions, or Google Cloud Functions.

Serverless Computing

Common Use Cases

- REST APIs and microservices
- Real-time file or data processing
- Scheduled tasks (cron jobs)
- Chatbots and notification systems
- Backend for mobile/web apps

Serverless Computing

Popular Serverless Platforms

- AWS Lambda
- Azure Functions
- Google Cloud Functions
- Cloudflare Workers
- Netlify Functions

Serverless Computing

Advantages

- No infrastructure management
- Cost-efficient for intermittent workloads
- Fast deployment and iteration

Serverless Computing

Disadvantages

- Cold start latency (initial delay when functions are idle)
- Limited execution time and memory
- Vendor lock-in risks

Microservices

Microservices

Description

- Microservices is an architectural style that structures an application as a collection of small, autonomous services, each responsible for a specific business capability.
- These services are independently deployable, loosely coupled, and communicate over a network, typically using lightweight protocols like HTTP or messaging queues.
- Composable: compare to Unix/Linux commands and pipes
- Components: compare to audio equipment
    - Radio, turntable, CD player, MP3 player, amplifier, speakers

Microservices

Core Characteristics - 1

- Single Responsibility
  - Each service focuses on a specific business function (e.g., user management, billing, inventory).
- Independent Deployment
  - Services can be updated, deployed, and scaled independently.
- Decentralized Data Management
  - Each service often manages its own database, avoiding shared data stores.

Microservices

Core Characteristics - 2

- Technology Diversity
  - Teams can use different programming languages, frameworks, or databases for different services.
- Resilience and Fault Isolation
  - Failures in one service are less likely to bring down the entire system.

Microservices

Typical Microservices Architecture

- API Gateway: Entry point that routes requests to appropriate services.
- Services: Independently running components (e.g., Auth Service, Product Service).
- Databases: Each service may have its own database.
- Communication: Often via REST, gRPC, or messaging systems like Kafka or RabbitMQ.

Microservices

Advantages

- Scalability: Scale services independently based on demand.
- Flexibility: Use the best tools for each service.
- Faster Development: Small teams can work in parallel.
    - “Two pizza” teams.
- Resilience: Isolated failures reduce system-wide impact.

Microservices

Disadvantages

- Complexity: More moving parts to manage.
- Operational Overhead: Requires service discovery, monitoring, logging, etc.
- Data Consistency: Harder to maintain consistency across services.
- Deployment: Requires orchestration tools like Kubernetes.

Comparisons

Aspect	Traditional Monolith	Modular Monolith	Microservices
Code Organization	Often tangled and tightly coupled	Clearly separated modules with strict boundaries	Independent services
Deployment	Single unit	Single unit	Independent units
Scalability	Whole app	Whole app (but easier to isolate bottlenecks)	Per service
Maintainability	Harder as app grows	Easier due to modular structure	Easier, but more complex to manage

Comparisons

Aspect	Traditional Monolith	Modular Monolith	Microservices
Testing	Simple but can be slow	Modular testing possible	Requires integration testing across services
Team Autonomy	Low	Moderate (teams can own modules)	High (teams own services)
Operational Complexity	Low	Low to moderate	High (networking, orchestration, observability)

Microservices

The Good, the Bad, and the Ugly

- Good
  - Enforced modularity around business capabilities, simple deployable components
- Bad
  - Distributed systems have complex interactions and communications
  - Door Dash has 500 services and requires 100 to place an order; 1000 RPCs per order!
- Ugly
  - Database inconsistency

Microservices

Ideal

- Enforced modularity
- Independent teams
- Independent deployment
- Fast (function calls rather than network RPC)
- Database consistency

GemStone/S 64 Bit

GemStone

Can be a Classic Monolith

- One customer has 141 thousand classes, 2.5 million methods, and 35 million lines of code.
  - 152 thousand tests
- Common to update all code at once.
- Module boundaries typically depend on developer discipline.
- Accessibility of code and data makes it tempting to break module boundaries.

GemStone

Zero-downtime Deployment

- In Smalltalk, classes and methods are objects (code is data).
- In GemStone, data is modified in a transaction and immediately available to other sessions.
- After a session abort/commit, the next message send will see the new code.

GemStone

User Isolation

- Each login user has their own object graph root.
- Each object has a security policy that specifies access based on owner, group, world.
- Each login user can define code that is executable but not directly visible to other users.
- Each “module” could be loaded into a designated user space.
- This would enforce module boundaries but preserve performance of function call.

Microservices in GemStone

- Enforced modularity
- Independent teams
- Independent deployment
- Fast (function calls rather than network RPC)
- Database consistency

Questions?

James.Foster@GemTalkSystems.com