# Virtualization of ArcGIS Services:

# Delivering Massively Scalable GIS Services with ArcGIS and Grid-based Technology

**Scott Crawford, Solution Architect, Appistry Inc.**

FedUC 2008

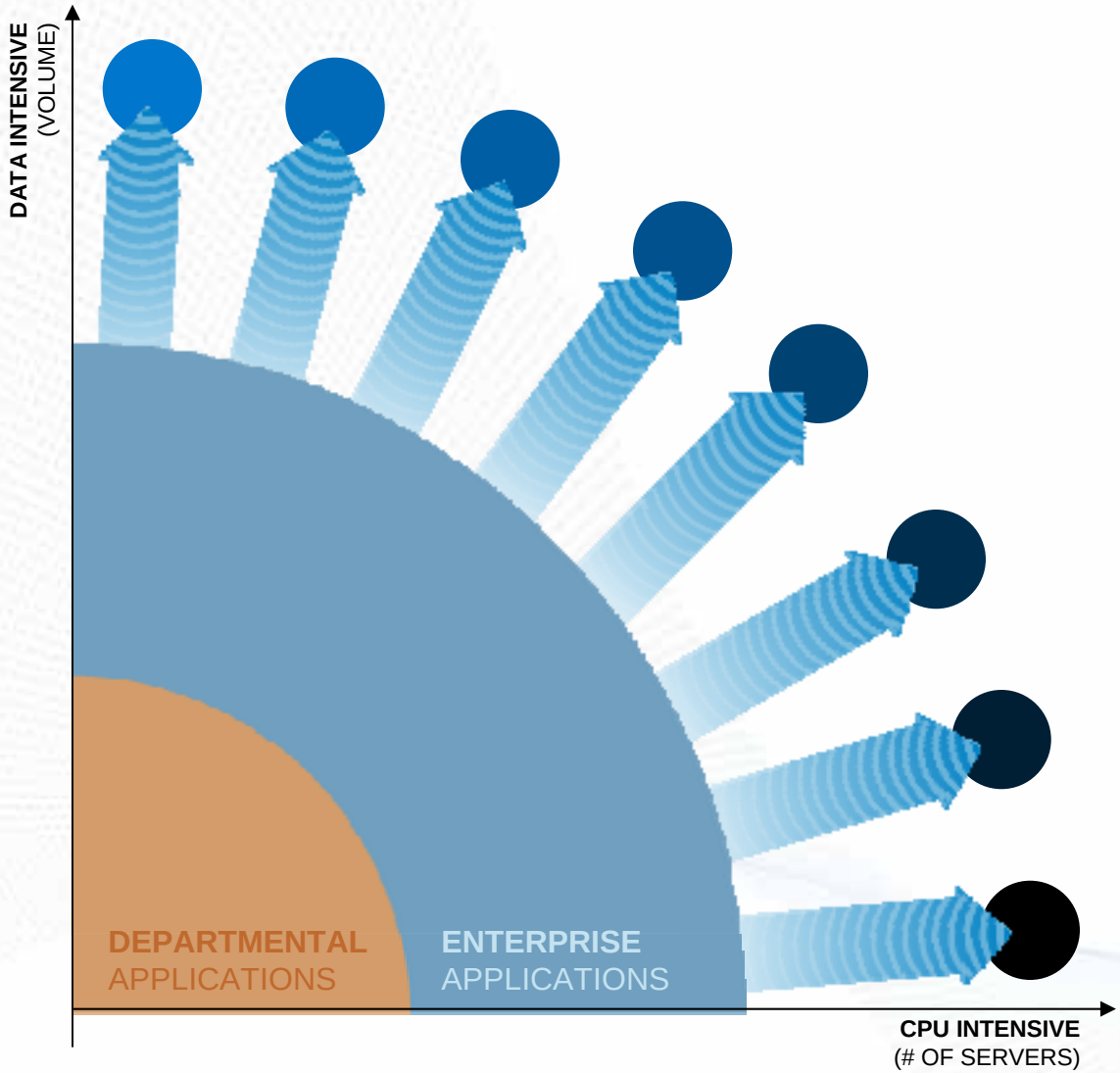
*"Fresh, radical and powerful technology aimed at meeting the needs of highly demanding business problems."*

– **Massimo Pezzini**, Gartner

# Agenda

- New challenges for a GIS-driven world
- Traditional Approaches/Challenges
- A new breed of solution
- Proof Points/Considerations
- Future Work

# Abundant GIS data driving new requirements




- Data sources
- Sophisticated analytics
- Internet distribution
- New business models
- System to System (SOA)
- Customer Expectations
- Mobile Access
- Constant change

# Traditional Approaches/Challenges:

## Possible Solutions

- ArcGIS Server Scale-out with SOM/SOC
- Traditional GRID technologies

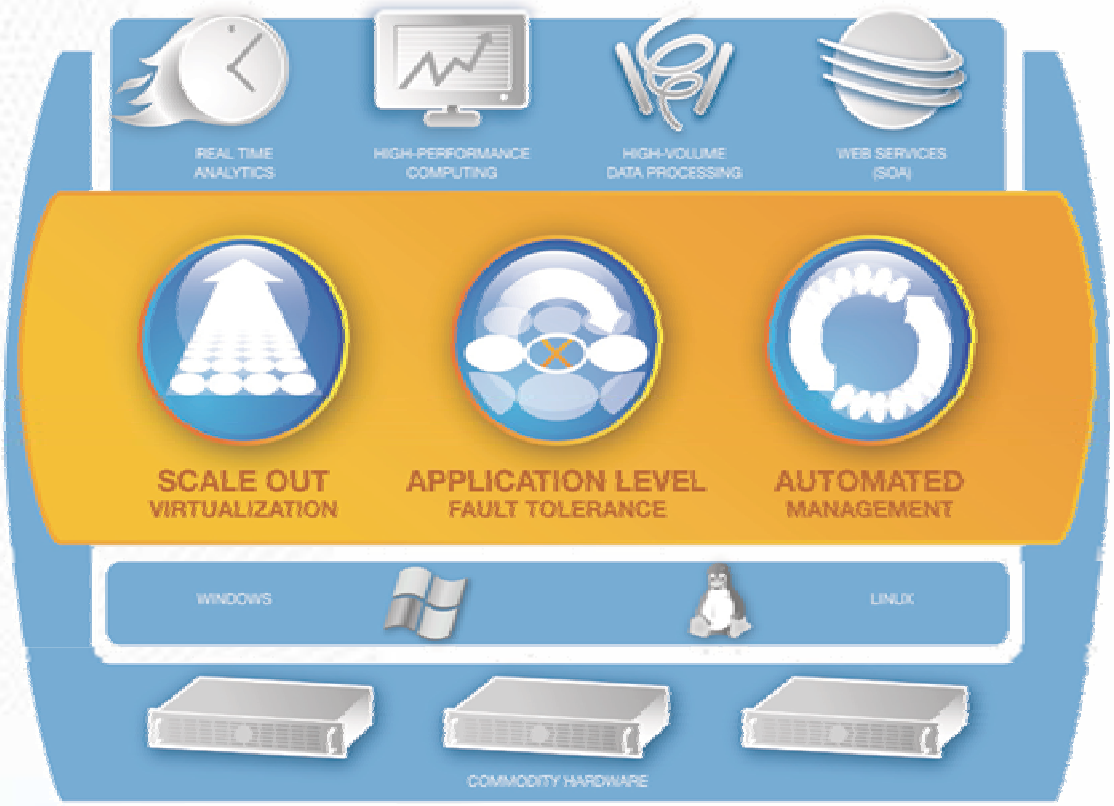
## Challenges

- Development & Operational Complexity
- Potentially undesirable behavior in error conditions
  - System Overload
  - Server Failures
- ArcGIS Server tuned for interactive jobs
- GRID tuned for batch jobs… no determinism

# What we need/want

- Effortless scale, on-demand
  - Self configuring apps
- Decreased time-to-market for new capabilities
  - Allow developers to focus on GIS function
- Complete reliability
  - Every service invocation runs
- Commoditized infrastructure
  - 1 U's or blades
- Unified deployment model across technology platforms
  - Linux, Windows, Java, .Net
- Simplified operations

# Appistry EAF




**Appistry Enterprise Application Fabric**

**A grid-based application platform that dramatically simplifies the development and deployment of agile applications for high-volume data/transaction processing**

# Massive scale out of ESRI ArcGIS functionality

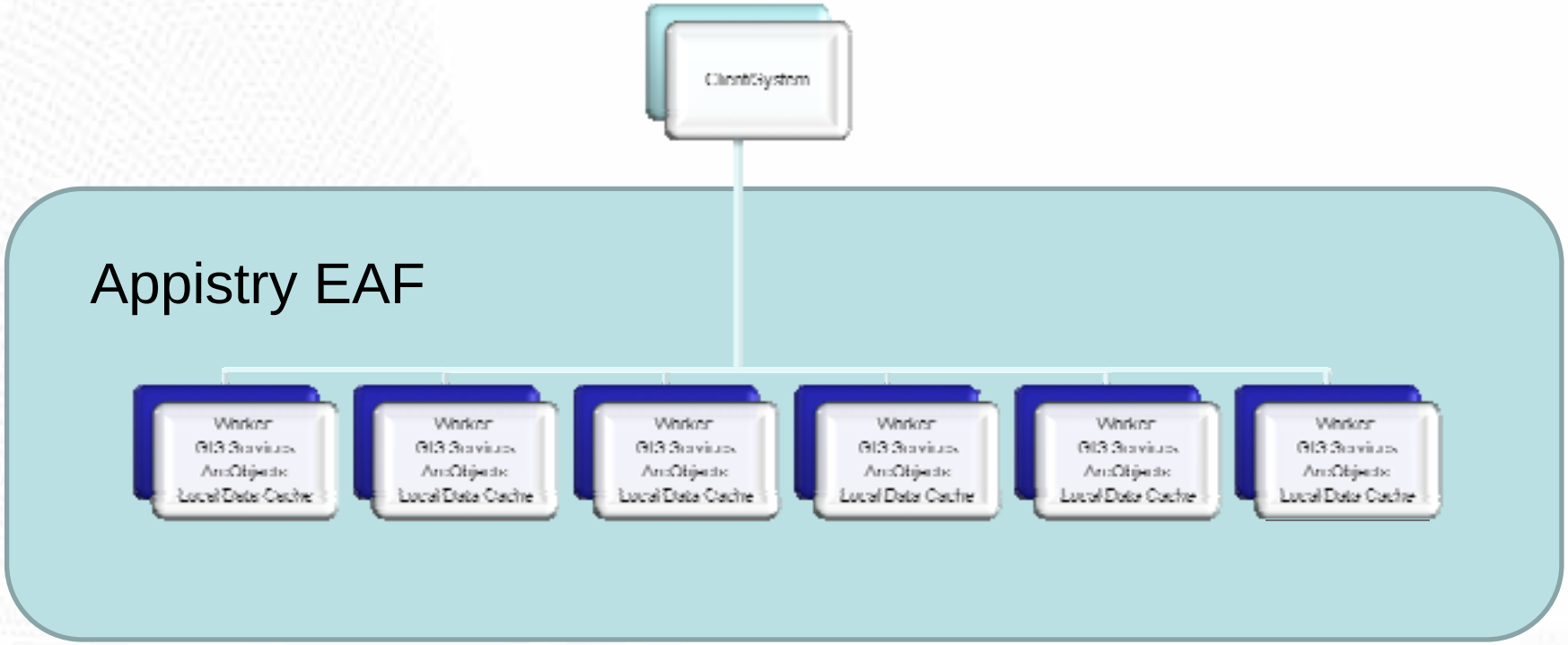
## Need

- ArcGIS Functionality requiring BOTH scale AND time sensitivity
  - Services used by many simultaneous users
  - High Volume GIS Data Processing
- Undetermined need for compute power
  - SOA service endpoint deployment
  - Rapid response to emergencies or new missions

## Solution

- Appistry EAF based GIS solution
- Hosting ESRI ArcObjects on an application fabric
- Transparently Scalable to 100's of processors
- Servers are self provisioned, self healing

# GIS Fabric Deployment




- Scales to 100's of processors easily
- EAF provides reliability of GIS services
- Windows or Linux

# Proof Points/Considerations

- Common coarse grained GIS services
  - High Volume Batch Geocoding
  - Linearly Scalable PDF Map Generation
  - Geoprocessing Jobs - Network Analyst
- Java/.Net services implemented using ArcGIS Engine API
- Data centrally managed, cached locally
- Processing within service atomic to single node

# Batch Geocoding

- Java service implemented using ArcGIS Engine API
  - From sample desktop Java app to scalable EAF service in 1 week
- Utilized 16 Dell PowerEdge servers:
  - Two Dual-Core Intel Xeons (4 cores per server)
  - 2 GB RAM
- Sustained > 50,000 addresses/second processed
- Complete reliablity… passed “pull-the-plug” test

- Locators deployed to each node
- State Level Locators used due to limited RAM
- One Runtime Engine instance stood up per core (Engine is single-threaded)

# PDF Map Generation

- .Net (VB.net and C#) service implemented using ArcGIS Engine API
  - Generates pdf or jpeg map zoomed to a feature (currently zip codes)
  - From sample desktop VB.net app to EAF service in < 2 days
- Utilizing 10 Aopen computers (mac-mini equivalent):
  - One Core2-duo processor (2 cores per server)
  - 1 GB RAM
- Sustained ~ 200 pdf maps generated per minute
  - Linear scale
- Allows interactive jobs concurrent to batch jobs
- Complete reliablity… passed “pull-the-plug” test

# PDF Map Generation - Design

- Base/Source Map deployed to each node
  - All SDC data local in flat files
  - Maintained on central server
    - Image server
    - NAS – compute nodes pull new version as available
- One process container per core (two RTE instances per server)
  - Base map is preloaded in each container
- Geometry and symbol data can be serialized into request and added to map dynamically
- Client can do batch (any number of unique jobs up to 30,000 zips) as well as interactive (enter a zip in the GUI)

# Future Work

- Virtualize ArcGIS Server
  - Deploy server to each node
  - EAF to configure SOM/SOCs
  - EAF handles load distribution and failover of jobs
  - Map Tile Cache Generation (big batch job) likely candidate
- Web Tier on EAF
  - Fully scalable SOAP stack and JSP/Servlet capabilities
- Smart Data Caching/optimized interactive processing
  - EAF “task affinity” feature brings jobs to data
- Network Analyst capability and other geoprocessing tasks

# Follow-up

- Email to scott@appistry.com with any questions or for a copy of the slides.
- Check out www.appistry.com
- www.appistry.com/developers - new developer portal including free software for up to 5 servers
- Think Big!  What sorts of things could you do with unlimited GIS processing capabilities?

# Tech Backup Slides

*"Fresh, radical and powerful technology aimed at meeting the needs of highly demanding business problems."*

– **Massimo Pezzini**, Gartner

# Fabric Services

- Written as “Process Flows” – familiar and comfortable for developers
    - XML
- Each step or activity in the process flow is an entry point into an atomic “piece of work.” These are called “Tasks” in EAF application
    - User provided method/function
    - Any executable or OS script that runs to completion
- Process flow “state payload” flows between each task and back to client on completion
- Tasks can store state:
    - Within the Process Flow “state payload”
    - Database connectivity (ODBC/JDBC or Object Relational Mapping tools)
    - Fabric Accessible Memory (reliable in-memory storage maintained by EAF)
    - Shared Disk (NAS, etc.)

# For Developers

**Scale-out Virtualization**

Code once, scale on demand

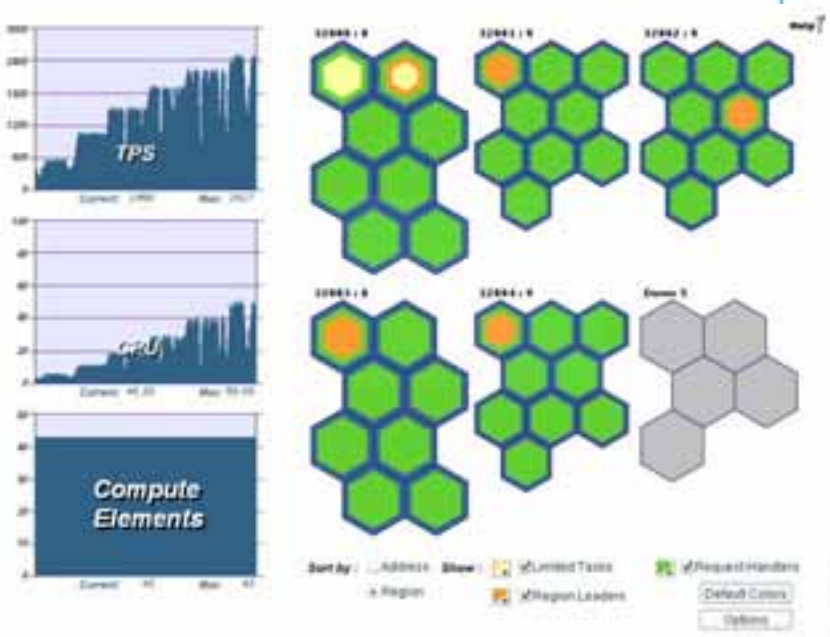
Appistry

- Power of a grid solution, with the simplicity of a single server.
- Developers can focus on domain and business function, not the distributed computing architecture.
  - Threading
  - Scalable State Replication
- Flexible architecture allows for deploying a wide range of applications.
  - Limited Tasks
  - Background Tasks
  - Exclusive Tasks
  - Task Affinity

# Scale-out Virtualization - How

**Scale-out Virtualization**

- Unified identity (IP address)
  - Each worker is capable of executing any service deployed to the fabric.
  - Developers access the Fabric as one thing by its address.
- Self-Discovery
  - Workers in the Fabric find each other and learn how to work together to execute services that have been deployed to them.
- Load Balancing
  - Work is distributed within the Fabric based on:
    - Relative capability of each box
    - Current load
    - Past performance
- Self-Organization
  - Workers optimize their socket connections to isolate network traffic to small groups of workers with the lowest latency between them.



# For Developers

**Automated Management**

## Flexible Deployment

- Single action to deploy to “N” workers for test
- Use Fabrics to provide logical partitioning of Testing Environments
- Easily/quickly repurpose machines for various testing activities
    - Move workers between test “levels”
    - Use all available test machines for large, realistic “user acceptance testing”

# For Administrators

**Automated Management**

Manage Fabric as Single Server

- No touch server provisioning
  - EAF interoperates with imaging solution to provision OS and EAF rpm.
  - EAF discovers other workers and gets “autoprovisioned” with user apps and config
- Rolling Updates
  - Single administrative command to deploy apps and updates
    - Control # of workers offline at any time
    - Possible to roll new apps or versions while under load
    - Can force a rolling PXE boot for OS re-image
  - All deployments and config changes automatically get to all workers in the fabric
- Simple configuration
  - All fabric configuration done via a single, simple fabric.cfg file
- Simple installation
  - Silent install on Windows, single RPM on Linux

Automated Management

# Automated Management - How

- Collective intelligence to incorporate updates in a controlled way
  - Temporal role of one machine to manage update
    - Another worker will take over on failure
  - Workers volunteer to cycle
- Graceful Cycle
  - Stop accepting new work
  - Finish all work committed to already
  - Cycle processes
- Self healing for version skew
  - Workers constantly comparing local “manifest” with peers
  - Out-of-date pieces automatically updated to latest
- Force PXE boot for OS upgrade

# For Developers

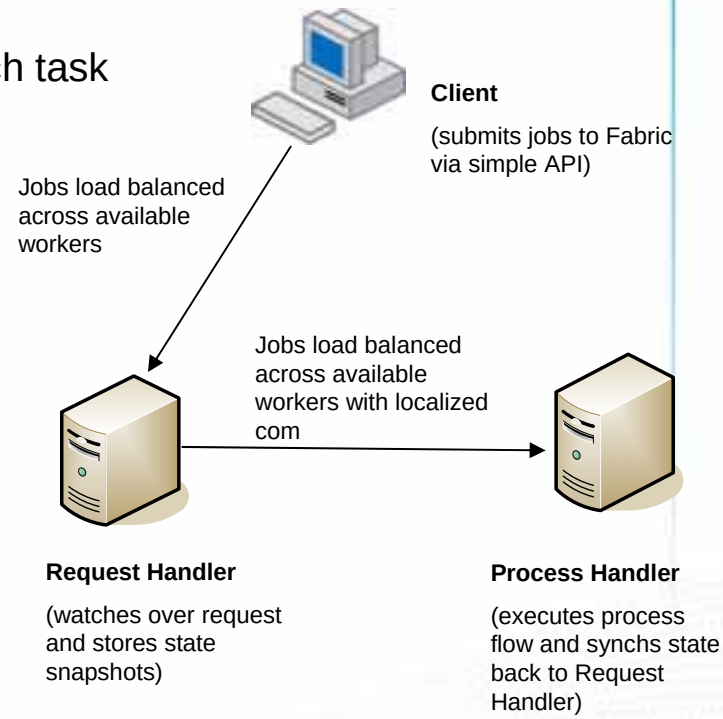
**Application-level Fault Tolerance**

Just Count on It

- Enjoy complete reliability by simply specifying action to take on failure
- Let the Fabric take care of:
  - Keeping track of available servers and current workload
  - Distributing work to available servers
  - Watching over servers to detect failures
  - Watching over tasks to ensure they are completed
  - Moving work from failed server to backup server (and re-loadbalancing)
  - Keeping data consistent when fail-over occurs

# Application-Level Fault Tolerance - How

- Workers pair up dynamically to execute a Process Flow Invocation
  - One worker does the work
    - Executes the Process Flow state machine
      - Finds / delegates to worker to execute each task
  - One worker watches over the work
    - Keeps track of progress
    - Holds state snapshots
- On failure, the remaining worker does the following:
  - "Re-Load balances" the work to another worker
  - Continues on to completion
  - Gets results to the client



# Typical Implementation Process

- Phase 1 – Fabric Enabled Services
  - Deploy current service as single task (4 hours)
    - Supports scale-out, and automated retries
    - Simple task wrapper to marshal input/output data
    - Serialize complex data into single input
  - Build service “remote interface” using client API (2 hours)
    - Minimize changes required by upstream systems
  - Package existing service and dependencies into Fabric Deployment Package (2 hours)
    - Transparent scale by adding new workers
- Phase 2 – Fine Grained Fail-over (as needed)
  - Break service into multiple tasks
  - Map top–level control flow to Process Flow definition
  - Build “compensating” states to ensure external data consistency
    - Services can fail-over with minimal “re-execution”

# API Samples

# Fabric Client – Execute a Service

```
import com.appistry.fabric.*;

public class SimpleOrderClientSync
{
	public static void main(String[] args)
	{
		String fabricIP = "239.255.0.1";
		int fabricPort = 31000;

		// create an instance of the Fabric object
		Fabric fabric = new Fabric( fabricIP, fabricPort );

		try
		{
			// create a request object with application name and process flow name
			FabricRequest request = new FabricRequest( "order_app", "order_item“ );

			// initialize the data for the fabric request
			request.put( "ITEM", "item name" );
			request.put( "QUANTITY", 2L );
			request.put( "ORDER_NUMBER", "123456" );
			request.put( "PURCHASE_AMOUNT", 100.0 );
			request.put( "CC_NUMBER", "4012881188888888" );
			request.put( "EXPIRATION_DATE", "1204" );

			fabric.execute( request );
			processResults( request );
			    }
		catch ( FabricClientException e )
		{
			e.printStackTrace();
		}
	}
}
```

# Fabric Process Flows – Service Control Flow

```
<process name='order_item' default_retries='2' default_timeout=‘10'>
        <state name='start'>
                <task name='itemLookupHold‘ snapshot=‘true’/>
                <arc task-status='complete' task-result=‘HELD' next-state=‘purchase'/>
                <arc task-status='complete' task-result='OUT_OF_STOCK' next-state='finish'/>
        </state>
        <state name='purchase' snapshot='true'>
                <task name='itemPurchase' retries=‘0' timeout='60'/>
                <arc task-status='complete' task-result='PURCHASED' next-state='finish'/>
                <arc task-status='complete' task-result='PURCHASE-DENIED' next-state='hold-release'/>
                <arc task-status=‘not-completed' next-state=‘rollback-purchase'/>
        </state>
        <state name='hold-release' snapshot='true'>
                <task name='itemReleaseHold'/>
                <arc task-status='complete' next-state=‘finish'/>
        </state>
        <state name=‘rollback-purchase' snapshot='true'>
                <task name=‘rollbackPurchase'/>
                <arc task-status='complete' next-state=‘purchase'/>
        </state>
</process>
```

# Fabric Tasks – Service Processing

## Plain Java

```
public class ItemPurchaseTask
{
        String _approvalCode;

       String _result;

        public void itemPurchase(String creditCardNumber,
                                 String expirationDate,
                                 String amount,
                                 String orderNumber) throws Exception
        {

            _approvalCode = creditCardPurchase(creditCardNumber,
                                               expirationDate,
                                               amount,
                                               orderNumber);

            if (_approvalCode.equals(""))
                _result = "PURCHASE-DENIED";
            else
                _result = "PURCHASED";
        }
}
```

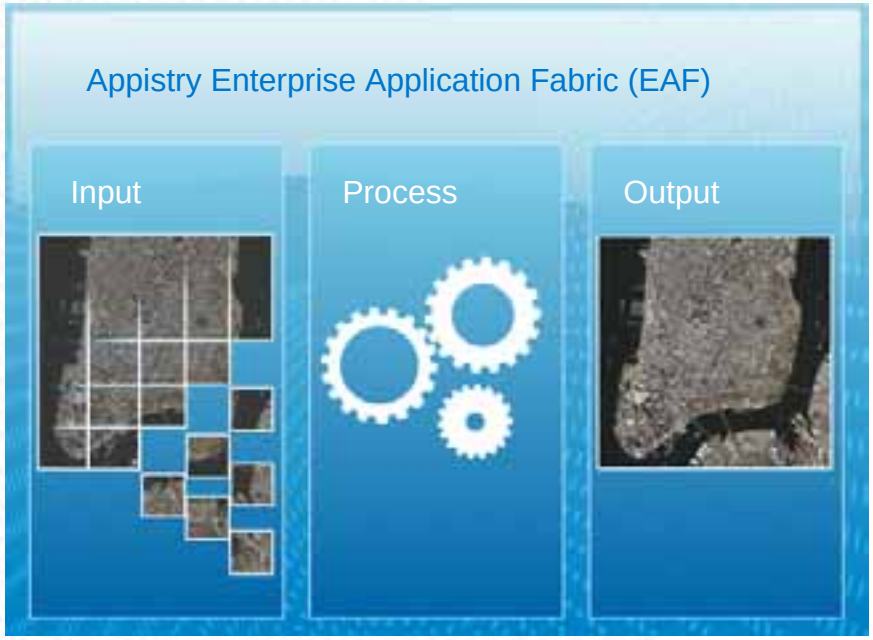
## Fabric Task

```
import com.appistry.task.Annotations.*;

public class ItemPurchaseTask
{
        @TaskField (type = FieldType.OUT)
        public String _approvalCode;

        @TaskResult
        public String _result;

        public void itemPurchase(@TaskParameter("CC_NUMBER") String creditCardNumber,
            @TaskParameter("EXPIRATION_DATE") String expirationDate,
            @TaskParameter("PURCHASE_AMOUNT") String amount,
            @TaskParameter("ORDER_NUMBER") String orderNumber) throws Exception
        {
            _approvalCode = creditCardPurchase(creditCardNumber,
                                               expirationDate,
                                               amount,
                                               orderNumber);

            if (_approvalCode.equals(""))
                _result = "PURCHASE-DENIED";
            else
                _result = "PURCHASED";
        }
}
```

# GeoEye Image Processing Details

# Case Study:

## Decreasing Development Complexity for a High-Volume Data Processing Application




- ***Input:*** Raw satellite image is retrieved from SAN and broken up into tiles
- ***Process:*** Tiles are processed using proprietary algorithms for sharpening, geo-correction, etc.
- ***Output:*** Tiles are reassembled and stored back in SAN

**The leading provider of satellite imagery for government and commercial applications, GeoEye is building its next-generation image processing applications on Appistry EAF.**

*Value Proposition:*

Because Appistry EAF allows GeoEye to deploy its applications on commodity infrastructure, the company cuts capital costs dramatically compared to “big iron” approaches

Appistry EAF allows GeoEye developers to focus on image processing, eliminating the need to grapple with arcane distributed computing concepts

*Key Data Points:*

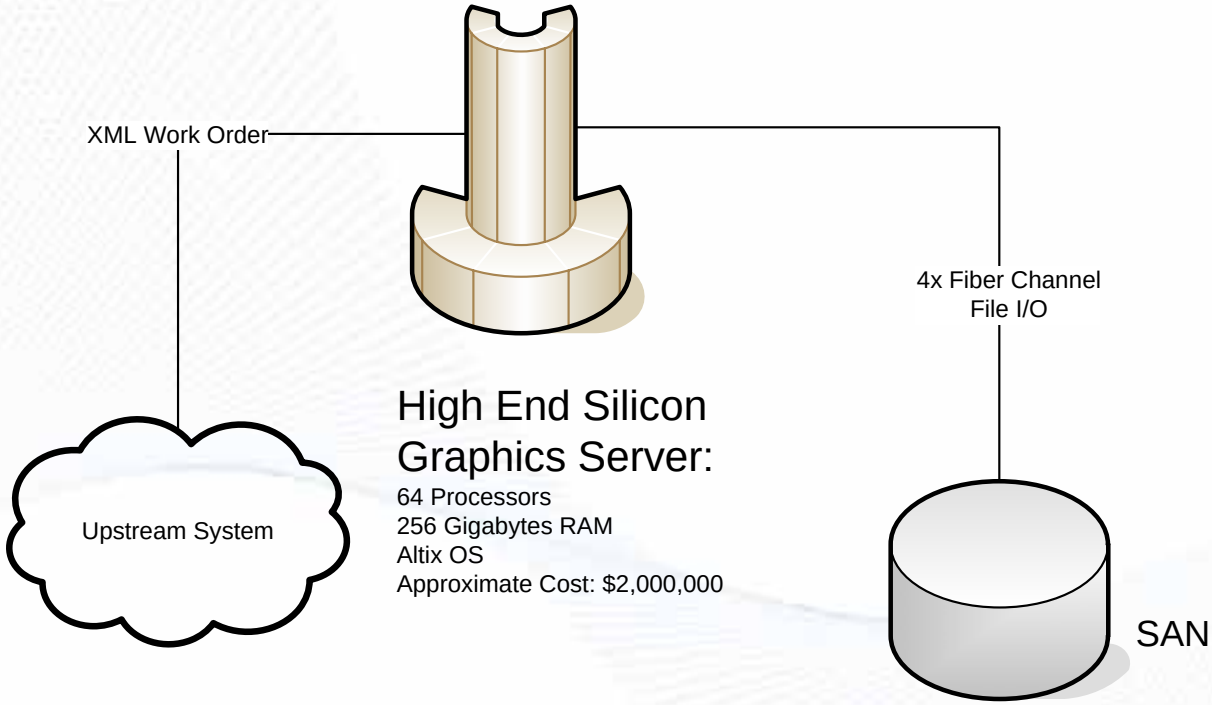
Company’s imaging applications process in excess of 5 TB of satellite imagery per day

Customer requirements dictate that each image be processed in under 6 minutes

# GeoEye:
## Traditional Image Processing Design

Automated Image Processing System:

Receive XML WorkOrder
Read Input Images from SAN ~17 GB
Sharpen Image
Geocorrect Image
Compress Image
Write Results to SAN
Generate XML WorkerOrderResponse




High End Silicon Graphics Server:

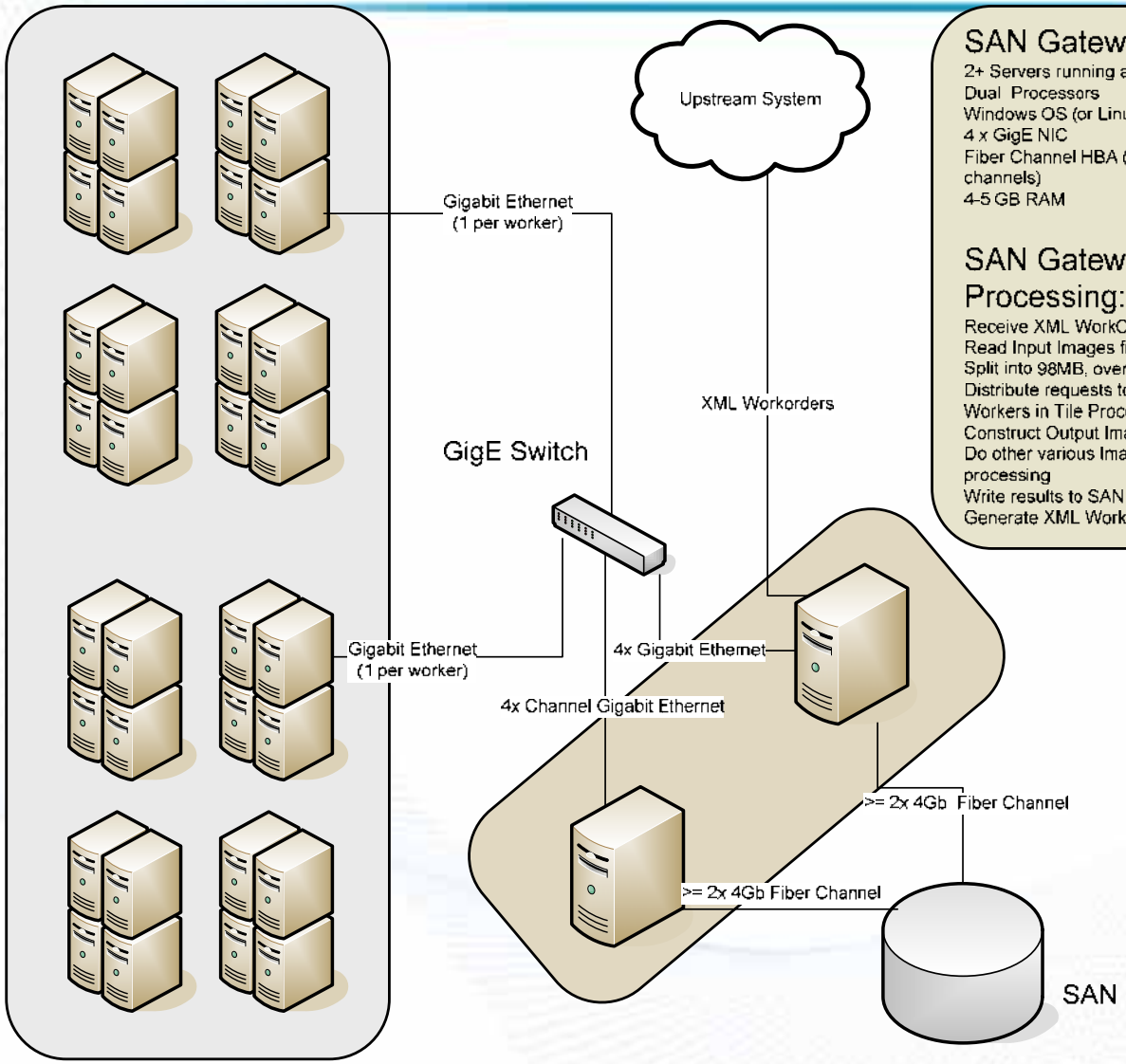
64 Processors
256 Gigabytes RAM
Altix OS
Approximate Cost: $2,000,000

# GeoEye:
## Fabric-Based Image Processing Architecture

Appistry

### Tile Processing Fabric Layer:
50-100 Workers (as needed)
Pentium P4 3 Ghz Typical
1 GB RAM
Linux OS
GigE NIC

### Tile Processing:
Load Tile from SAN Gateway Layer
Sharpen
Resample (Geocorrection)
Other Tile Level Processing
Compress (Jpeg 2000)
Write Tile to SAN Gateway Layer
Can Scale Up Workers as Needed




### SAN Gateway Layer:
2+ Servers running as separate Fabric
Dual Processors
Windows OS (or Linux)
4 x GigE NIC
Fiber Channel HBA (minimum 4, 4 Gbs channels)
4-5 GB RAM

### SAN Gateway Processing:
Receive XML WorkOrder
Read Input Images from SAN ~17 GB
Split into 98MB, overlapping tiles
Distribute requests to process Tiles to Workers in Tile Processing Fabric
Construct Output Images
Do other various Image level processing
Write results to SAN
Generate XML WorkerorderResponse

# Results:
## Lowered Acquisition/Op Costs, SLA Met

- Savings of over $1.2M using standard hardware
  - 60 Worker Fabric: 2.8 GHz Pentium 4 with 512 MB ram, gigabit Ethernet
- Handling Massive I/O
  - Input: 30 GB raw imagery, “tiled” by fabric into 50 MB chunks
    Output: 3 GB compressed imagery -- corrected & sharpened
- SLA met:
  - Elapsed Time: < 6 min (4.5 minutes)
- Demonstrated linear throughput scalability
  - reducing image processing latency as additional workers are added