1. You Oversell Agents

• Agents are not a magic bullet.
• They do not make the impossible possible.
• Agents may make it easier to solve certain classes of problems...but they do not make the impossible possible.

• Agent-oriented development requires new agent-oriented techniques.
• No evidence that any system developed using agent-oriented development could not have been built just as easily if you can’t do it with agents, you probably can’t do it with ordinary software; you probably can’t do it with agents.

2. Seven categories of pitfalls.

- Micro (agent) level:
- Analytical and design:
- Conceptual:
- Management:
- Political:

- Seven categories:
- Identifies key pitfalls.

We now consider pragmatics of AO software projects.

Lots of single and multi-agent projects...but 1 pitfall of agent development
1.2 You Get Religious

• Agents have been used in a wide range of applications, but they are not a universal solution.

Other projects appear to be going well. ("We have gained from it.

The lesson: understand your reasons for attempting agents.)

Other form of dogma: believing in your agent solution to every problem.

In summary: danger of believing that agents are the right solution to every problem.

Given a problem for which an agent and a non-agent approach appear equally good, prefer non-agent solutions.

For many applications, conventional software (e.g., OO) are more appropriate.

Agents have been used in a wide range of applications, but they are not a universal solution.

1.3 Don’t Know Why You Want Agents

• Agents = new technology = lots of hype?

Year 2000: "Agents will generate US $2.6 billion in revenue by the end of the year.

Managerial reaction: we can get 10% of that.

Managers often propose agent projects without having clear idea about what "having agents" will buy them.

No business plan for the project.

• Solutions vendor?
• Technology vendor?
• Pure research?

The lesson: understand your reasons for attempting agents.

Don’t equate agents and AI.

1. Don’t equate agents and AI.
1.4 Don't Know What Agents Are Good For

- Having developed some agent technology, you search for an application to use them.
- Putting the cart before the horse!
- Leads to mismatches/disillusionment - Prototyping is easy (particularly with nice GUI tools)
- The lesson: be sure you understand how and where your new technology may be most usefully applied. Do not attempt to apply it to arbitrary problems.

1.5 Generic Solutions to 1-off Problems

- Diseased solution and building testbeds
- Re-use is difficult to attain unless development is undertaken for a close-knit range of problems with similar characteristics.
- Re-use is difficult to attain unless development is undertaken for a close-knit range of problems with similar characteristics.
- Re-use is difficult to attain unless development is undertaken for a close-knit range of problems with similar characteristics.
- Re-use is difficult to attain unless development is undertaken for a close-knit range of problems with similar characteristics.

1.6 Confuse Prototypes with Systems

- Prototypes are easy - Having developed some agent technology, you search for an application to use them.
- Putting the cart before the horse!
- Leads to mismatches/disillusionment - Prototyping is easy (particularly with nice GUI tools)
- The lesson: be sure you understand how and where your new technology may be most usefully applied. Do not attempt to apply it to arbitrary problems.
Chapter 9: An Introduction to Multiagent Systems

Good reasons to believe that agents are useful ways of tackling some problems.

But these arguments largely untested in practice.

Useful developments in software engineering:

Abstractions.

Agents are another abstraction.

Label "BDI" now been applied to WWW pages/Perl scripts.

Oursystemisa BDI system “implies that this is like being a computer with 64MB memory: a
quantifiable property, with measurable associated benefits.

- Logic of practical reasoning (Rao & Georgeff).
- Serious applications (NASA, ...).

Developing software.

- The idea of an agent is extremely intuitive.
- Project plans focus on the agent bits.
- Encourages developers to forget they are
  experimenting.
- Not tried and trusted techniques
  forgotten.
- Moresoftwareengineering(requirements
  analysis, specification, design, verification, testing) is
  forgotten.

1.8 Confuse Buzzwords & Concepts

- Good example: the belief-desire-intention(BDI)
  model.
- Theory of human practical reasoning (Bratman et
  al).
- Project architectures (PARS, DMARS,...).

1.9 Forget it's Software

- The AI & party syndrome: everyone has an opinion.
- Encourages developers to believe in their
  understanding concepts when they do not.
- The idea of an agent is extremely intuitive.
- Developing any agent system is essentially
  experimentation.
- Developing software is software engineering.
- Good software engineering (specification, design, verification, testing) is
  essential.

1.9 Confuse Buzzwords & Concepts

- Good example: the belief-desire-intention(BDI)
  model.
- Theory of human practical reasoning (Bratman et
  al).
- Project architectures (PARS, DMARS,...).

1.9 Forget it's Software

- The AI & party syndrome: everyone has an opinion.
- Encourages developers to believe in their
  understanding concepts when they do not.
- The idea of an agent is extremely intuitive.
- Developing any agent system is essentially
  experimentation.
- Developing software is software engineering.
- Good software engineering (specification, design, verification, testing) is
  essential.
Result foregone conclusion: project flounders, not because agent problems, but because basic software engineering ignored.

Frequent justification: software engineering for agent systems is none-existent.

But almost any principled software development technique is better than none.

Forget its distributed nature.

Distributed systems = one of the most complex classes of computer system to design and implement.

Therefore important that conventional technologies and techniques are exploited wherever possible.

The raising bread model of Winston.

In any agent system, percentage of the system that is agent-specific is comparatively small.

1.10 Don't Exploit Related Technology

In any agents system, percentage of the system that is agent-specific is comparatively small.

Therefore important that conventional technologies and techniques are exploited wherever possible.

Don't reinvent the wheel. (Yet another communication framework.)

Exploitation of related technology:

- speeds up development;
- avoids re-inventing wheel;
- makes use of DS expertise.

Example: CORBA.

Recognise distributed systems problems:

- a typical distributed system will be more complex than a system is agent-based;
- problems of distribution do not go away, just because multi-agent systems are distributed;
- multi-agent systems tend to be distributed;
- classes of computer system to design and implement.

Forget its distributed nature.
1.11 Don’t exploit concurrency

- Many ways of cutting up any problem. Examples: decompose along functional, organisational, physical, or resource-related lines.
- One of the most obvious features of a poor design is that the amount of concurrent problem-solving is small or even non-existent.
- Serial processing in distributed systems.

1.12 Want your own architecture

- Agent architectures: designs for building agents. Many agent architectures have been proposed over the years.
- Great temptation to imagine you need your own.
- Driving forces behind this belief:
  - “not designed here” mindset;
  - Intellectual property.
- Problems:
  - Architecture development takes years.
  - No clear payback.

Recommendation: buy one, take one off the shelf, or:

- If you don’t exploit concurrency, why have an agent?
1.13 Think Your Architecture is Generic
• If you do develop an architecture, resist believing such features are essential in your agent system.
  • Resist the temptation to believe such features are useful first strategy.
  • Resist the temptation to believe such features are so dangerous.
• Different architectures good for different problems.
• Any architecture that is truly generic is by definition not an architecture.
  • Any architecture that is truly generic is by definition not an architecture.
• If you develop an architecture, resist believing it is generic.
• If you have developed an architecture, resist applying it to problems with similar characteristics.
• Only apply the architecture to problems with similar problem understanding why it succeeded with that particular problem. Therefore, be applied to some particular problem.

1.14 Use Too Much AI
• Temptation to focus on the agentspecific aspects.
  • Result: an agent framework too overburdened with experimental AI techniques to be usable.
  • Result: an agent framework too overburdened with experimental AI techniques to be usable.
• Fuelled by “feature envy,” where one reads about agents that have the ability to learn, plan, talk, sing, dance... Fueled by “feature envy,” where one reads about agents that have the ability to learn, plan, talk, sing, dance... Fueled by “feature envy,” where one reads about agents that have the ability to learn, plan, talk, sing, dance... Fueled by “feature envy,” where one reads about agents that have the ability to learn, plan, talk, sing, dance...
• Resist the temptation to focus on the agentspecific aspects.

What Etzioni calls “useful first strategy.”
1.15 Not Enough AI

- Don’t call your on-off switch an agent!
- Be realistic: it is becoming common to find everyday distributed systems referred to as multi-agent systems.
- Another example: referring to WWW pages as “agents”.
- Be realistic: it is becoming common to find everyday distributed systems referred to as multi-agent systems.
- Do not call on-off switches at agents.
- Problems:
  - Another common example: referring to WWW pages.
  - Be realistic: it is becoming common to find everyday distributed systems referred to as multi-agent systems.

1.16 See agents everywhere

- “Pure” A-O system = everything = an agent.
- More than 10 agents = big system.
- Choose the right grain size.
- Inappropriate.
- Naively viewing everything as an agent is inappropriate.

1.17 Too Many Agents

- Agents don’t have to be complex to generate complex behaviour.
- Large number of agents:
  - Emergent functionality;
  - Chaotic behaviour;
  - Too Many Agents;

Lessons:
- Keep protocol simple.
- Keep interactions to a minimum.
- Large number of agents: behaviour.
- Agents don’t have to be complex to generate complex behaviour.

- Lessons:
  - Don’t call your on-off switch an agent!
  - Be realistic: it is becoming common to find everyday distributed systems referred to as multi-agent systems.
  - Do not call on-off switches at agents.
1.18 Too few agents

- Some designers imagine a separate agent for every possible task.
- Others don’t recognize the value of a multi-agent approach at all.
- One “all powerful” agent.
- There are no widely-used software platforms for developing agent systems.
- Building and managing a system is more difficult to design, implement, test, debug, and manage.

A dangerous fallacy: distributed systems are an order
of magnitude more difficult to design, implement, test, debug, and manage.

A tendency to assume that results obtained with
simulation distribution will immediately scale up to real
highweight processes in C, or Java threads.

Every multi-agent system stars like on a single
computer.

1.20 System is anarchic

- Cannot simply bundle a group of agents together.
- Cannot assume simulated with real parallelism.

For large systems, or for systems in which the society
is supposed to act with some commonality of
purpose, this is particularly true.

A crucial mistake is to assume that results obtained with
simulated distribution will immediately scale up to real
highweight processes in C, or Java threads.

1.19 Implementing infrastructure

- Long road to develop a platform.
- By the time this is developed, project resources gone
- The result: everyone builds their own.
- Infrastructure required to create a multi-agent system.
- Such platforms would provide all the basic
infrastructure required to create a multi-agent system.
- In infrastructure required to create a multi-agent system.
- There are no widely-used software platforms for
developing agent systems.
- Failures software engineering test of coherence.
- Result is like the OO program with T class.
- One “all powerful” agent.
- Others don’t recognize value of a multi-agent
system.

1.18 Too few agents

- Some designers imagine a separate agent for every
agent.
- Failures software engineering test of coherence.
- Result is like the OO program with T class.
- One “all powerful” agent.
- Organizations structure (even in the form of formal
purpose) is particularly true.
- For large systems, or for systems in which the society
is supposed to act with some commonality of
purpose, this is particularly true.

A crucial mistake is to assume that results obtained with
simulated distribution will immediately scale up to real
highweight processes in C, or Java threads.

Every multi-agent system starts like on a single
computer.

1.20 System is anarchic

- Cannot simply bundle a group of agents together.
- Cannot assume simulated with real parallelism.

For large systems, or for systems in which the society
is supposed to act with some commonality of
purpose, this is particularly true.

A crucial mistake is to assume that results obtained with
simulated distribution will immediately scale up to real
highweight processes in C, or Java threads.

Every multi-agent system starts like on a single
computer.

1.20 System is anarchic

- Cannot simply bundle a group of agents together.
- Cannot assume simulated with real parallelism.

For large systems, or for systems in which the society
is supposed to act with some commonality of
purpose, this is particularly true.
Many practical problems in building distributed systems, from mundane to research level.

- Centralised control is not possible.
- With simulated distribution, there is the possibility of systems from mundane to research level.

1.23 Ignore de facto standards.

- There are no established agent standards.
- Centralised control is not possible.

- With simulated distribution, there is the possibility of systems from mundane to research level.

1.22 The tabula rasa

- Wrap each agent with an agent layer.
- They can be incorporated into an agent system by wrapping them with an agent layer.

- Legacy software components, functionally essential, but technologically obsolete software components, which cannot readily be rebuilt.
- Skip systems whose mission critical.
- When proposing a new software solution, essential to work with such components.

- Often, most important components of a software system will be legacy:
- "Black slate".
- When building systems using new technology, often in an assumption that it is necessary to start from a blank slate.

Examples:
- FIPA.
- KQML.
- HTML.
- CORBA.
- But there are some de facto standards.
- Simulated distribution and build all agent-specific components from scratch.
- Developers often believe they have no choice but to design and build all agent-specific components from scratch.

- When building systems using new technology, it is necessary to start from a blank slate.
- "Black slate".
- Often, most important components of a software system will be legacy:
- "Black slate".
- When proposing a new software solution, essential to work with such components.
Chapter 9
An Introduction to Multiagent Systems

2 Mobile Agents

Remote procedure calls (a) versus mobile agents (b):

Why mobile agents?
- low-bandwidth networks (hand-held PDAs, such as
  NEWTON);
- efficient use of network resources.
- heterogeneity of hosts and agents;
- security for hosts and agents;
- dynamic linking.

Why mobile agents?

There are many issues that need to be addressed
when building software tools that can support mobile
agents...

Security for Hosts

We do not want to execute foreign programs on our
machine, as this would present enormous security
risks:

- If the agent programming language supports pointers,
  then there is the danger of agents corrupting the
  address space of the host ⇒ many agent languages
don’t have pointers!
- UNIX-like access rights on host;
- safe libraries for access to file store, process space,
etc.

Some actions (e.g., sending mail) are harmless in
some circumstances, but dangerous in others — how
to tell?

Some actions (e.g., sending mail) are harmless in
some circumstances, but dangerous in others — how
to tell?
• some agent languages (e.g., TELESCRIPT) provide limits on the amount of e.g., memory & processor time that an agent can access.

• secure co-processors are a solution — have a physically separate processor on which the agent is run, such that the processor is in "quarantine" ("padded cell").

• This implies:
  - unless there is just one type of machine (Mac, PC, SPARC, ...), then we must provide facilities for executing the same agent on many different types of machine.

• interpreted languages imply reduction to machine code, which is clearly system dependent — reduced efficiency in a system of reduced efficiency.

• digital watermarks — rather like check digits.

• In order to ensure that an agent is not tampered with, conventional encryption techniques (e.g., PGP)

• An agent can be provided in terms by using an interpreted language (e.g., Python).

• The agent might be modified (subverted) in some way, without its owner's knowledge or approval.

• We then do not want to send our programs, so to agents have a right to privacy.

× Heterogeneity of hosts

× check digits

× Digital watermarks — rather like check digits.

× In order to ensure that an agent is not tampered with, conventional encryption techniques (e.g., PGP).

× An agent can be provided in terms by using an interpreted language (e.g., Python).

× The agent might be modified (subverted) in some way, without its owner's knowledge or approval.

× We then do not want to send our programs, so to agents have a right to privacy.
Wecandividemobileagentsintoatleastthreetypes:

– Autonomous;
– On-demand;
– ‘Activemail’-type.

**Autonomous Mobile Agents**

By autonomous, we mean agentsthatareable
todecidethemselves
wheretogo, when, and what
todowhentheygetthere (subjecttocertain
resource
costs). The best known example of such functionality is that
provided by the JAVA language, as embedded within
Java applets.

**On-Demand Mobility**

The idea here is that a host is only required to execute
an agent when it explicitly demands the agent.
Active-Mail Agents

- The idea here is to piggyback agent programs onto mail.
- The best-known example of this work is the mime extension to email, allowing Safe-Tcl scripts to be sent.
- When email is received, the agent is unpacked and sent.
- The idea here is to piggyback agent programs onto mail.

Places are virtual locations occupied by agents. A place may correspond to a single machine, or a family of machines.

2.1 Telescript

- TELESCRIPT was a language-based environment for constructing mobile agent systems.
- TELESCRIPT technology is the name given by General Magic to a family of concepts and technologies they have developed to underpin their products.
- TELESCRIPT technology is the name given by General Magic to a family of concepts and technologies they have developed to underpin their products.
- TELESCRIPT technology was a language-based environment for constructing mobile agent systems.
- TELESCRIPT technology is the name given by General Magic to a family of concepts and technologies they have developed to underpin their products.
Agents and places are executed by an engine.

- **Time** (measured in seconds).
- **Money** (measured in *teleclicks*, which correspond to real money).
- **Space:** What resources the agent can use.
- **Activity:** What the agent can do (e.g., limitations on travel).

**TELESCRIPT** agents have an associated *permil* which

- **TIME**.
- **Completion time**.

Agents can communicate with one another:

- If they occupy the same location, then they can connect across a network.
- If they occupy different places, then they can communicate with one another.

An engine is a kind of agent operating system — just as operating systems can limit the access agents corresponding to operating system processes.

Agents can communicate with one another:

- If they occupy the same place, then they can meet.

**Agents** have an associated *permit*, which specifies:

- What the agent can do (e.g., limitations on travel).
- What resources the agent can use.
- Lifetime (measured in seconds).
- Size (measured in bytes).
Engines continually monitor an agent's resource consumption and kill agents that exceed their limits.

Engines provide C/C++ links to other applications via application program interfaces (APIs).

Agents and places are programmed using the TELESCRIPT language:
- A rich set of primitives for building distributed applications,
- A powerful notion of agency;
- Support for agent-oriented languages—everything is an object;
- A process' class of which an agent and place are a subclass;
- Persistent agents;
- A semi-complete language for efficient execution;
TK is an X window based widget toolkit — it provides a complete set of widget sets.

TK also provides powerful facilities for interprocess communication, via the exchange of TCL scripts.

TK programs are called scripts.

TCL/TK can be embedded in an application, and can itself be extended.

As TCL programs are interpreted, they are very much easier to prototype and debug than compiled languages like C++. They also provide more powerful control constructs:...but this power comes at the expense of speed.

TCL/TK can be embedded — the interpreter itself is available as C++ code, which can be embedded in TCL programs are interpreted. They are very much easier to prototype and debug than compiled languages like C++. They also provide more powerful control constructs:...but this power comes at the expense of speed.

TCL/TK combined, make an attractive and simple to use GUI development tool: however, they have their limitations.

• TCL programs are called scripts.

TCL/TK can be embedded — the interpreter itself is available as C++ code, which can be embedded in an application, and can itself be extended.

As TCL programs are interpreted, they are very much easier to prototype and debug than compiled languages like C++. They also provide more powerful control constructs:...but this power comes at the expense of speed.

TCL/TK can be embedded — the interpreter itself is available as C++ code, which can be embedded in an application, and can itself be extended.

As TCL programs are interpreted, they are very much easier to prototype and debug than compiled languages like C++. They also provide more powerful control constructs:...but this power comes at the expense of speed.

TCL/TK combined, make an attractive and simple to use GUI development tool: however, they have their limitations.

• TCL/TK can be embedded — the interpreter itself is available as C++ code, which can be embedded in an application, and can itself be extended.

As TCL programs are interpreted, they are very much easier to prototype and debug than compiled languages like C++. They also provide more powerful control constructs:...but this power comes at the expense of speed.

TCL/TK combined, make an attractive and simple to use GUI development tool: however, they have their limitations.

• TCL programs are called scripts.

TCL/TK can be embedded — the interpreter itself is available as C++ code, which can be embedded in an application, and can itself be extended.

As TCL programs are interpreted, they are very much easier to prototype and debug than compiled languages like C++. They also provide more powerful control constructs:...but this power comes at the expense of speed.

TCL/TK combined, make an attractive and simple to use GUI development tool: however, they have their limitations.

• TCL programs are called scripts.

TCL/TK can be embedded — the interpreter itself is available as C++ code, which can be embedded in an application, and can itself be extended.

As TCL programs are interpreted, they are very much easier to prototype and debug than compiled languages like C++. They also provide more powerful control constructs:...but this power comes at the expense of speed.

TCL/TK combined, make an attractive and simple to use GUI development tool: however, they have their limitations.

• TCL programs are called scripts.

TCL/TK can be embedded — the interpreter itself is available as C++ code, which can be embedded in an application, and can itself be extended.

As TCL programs are interpreted, they are very much easier to prototype and debug than compiled languages like C++. They also provide more powerful control constructs:...but this power comes at the expense of speed.

TCL/TK combined, make an attractive and simple to use GUI development tool: however, they have their limitations.

• TCL programs are called scripts.

TCL/TK can be embedded — the interpreter itself is available as C++ code, which can be embedded in an application, and can itself be extended.

As TCL programs are interpreted, they are very much easier to prototype and debug than compiled languages like C++. They also provide more powerful control constructs:...but this power comes at the expense of speed.

TCL/TK combined, make an attractive and simple to use GUI development tool: however, they have their limitations.

• TCL programs are called scripts.

TCL/TK can be embedded — the interpreter itself is available as C++ code, which can be embedded in an application, and can itself be extended.

As TCL programs are interpreted, they are very much easier to prototype and debug than compiled languages like C++. They also provide more powerful control constructs:...but this power comes at the expense of speed.

TCL/TK combined, make an attractive and simple to use GUI development tool: however, they have their limitations.

• TCL programs are called scripts.

TCL/TK can be embedded — the interpreter itself is available as C++ code, which can be embedded in an application, and can itself be extended.

As TCL programs are interpreted, they are very much easier to prototype and debug than compiled languages like C++. They also provide more powerful control constructs:...but this power comes at the expense of speed.

TCL/TK combined, make an attractive and simple to use GUI development tool: however, they have their limitations.

• TCL programs are called scripts.

TCL/TK can be embedded — the interpreter itself is available as C++ code, which can be embedded in an application, and can itself be extended.

As TCL programs are interpreted, they are very much easier to prototype and debug than compiled languages like C++. They also provide more powerful control constructs:...but this power comes at the expense of speed.

TCL/TK combined, make an attractive and simple to use GUI development tool: however, they have their limitations.

• TCL programs are called scripts.
Chapter 9: An Introduction to Multiagent Systems

Where does the idea of an agent come in?

It is easy to build applications where TCL scripts are exchanged across a network, and executed on remote machines. Thus, TCL scripts become sort of agents. A key issue is safety. You don’t want to provide someone else’s script with the full access to your computer that an ordinary scripting language (e.g., C) provides. Thus TCL scripts are not intended as agent environments. But they are not used in their full extent as agent programming environments.

The core primitives may be used for building agent programming environments — the source code is free, stable, well-designed, and easily modeled. The core primitives include:

- TCL/TK provide a rich environment for building agent environments.
- The core primitives may be used for building agent programming environments.
- But they are not intended as agent environments.
- The core primitives may be used for building agent environments.

Summary:

- TCL/TK provide a rich environment for building agent environments.
- The core primitives may be used for building agent programming environments.
- But they are not intended as agent environments.
- The core primitives may be used for building agent environments.

An ordinary scripting language (e.g., C) provides:

- A key issue is safety. You don’t want to provide someone else’s script with the full access to your computer that an ordinary scripting language (e.g., C) provides.

So where does the idea of an agent come in?